University of Freiburg
Faculty of Engineering
Department of Computer Science

Master Thesis

Real-Time Movement Visualization of Public Transit Data

Patrick Brosi

March 2014

Supervisor

Prof. Dr. Hannah Bast

Period:
September 2013 - March 2014

Primary Reviewer:
Prof. Dr. Hannah Bast

Secondary Reviewer:
Prof. Dr. Christian Schindelhauer

Primary Supervisor:
Prof. Dr. Hannah Bast

Secondary Supervisor:
Dr. Sabine Storandt

Abstract

One of the primary challenges in visualizing movement in public transportation networks is handling the amount of data. This becomes particularly difficult when the visualization should be available in real-time and for arbitrary spatial parts of the dataset, for example to provide an interactive live map. The main task is then to reduce and crop the set of trajectories to the parts currently relevant for the visualization. In this thesis, we present an approach that combines static schedule data and real-time delay information into vehicle trajectories that are treated as piecewise linear functions from the time domain to the projected map-plane, allowing for fast interpolation. We introduce a highly efficient server that accepts temporal and spatial boundaries as an input and returns trajectory parts within these limits. We discuss the implementation details, show that our approach scales well and demonstrate that it is able to handle public transit datasets of entire countries. As a client, we introduce a thin vector-layer for map services that can display thousands of (time-lapsed) vehicle-movements at the same time. This thesis also briefly outlines an approach to extract vehicle trajectories from geospatial datasets.

Zusammenfassung

Eines der Hauptprobleme bei der Visualisierung von Fahrzeugbewegungen in Netzen des öffentlichen Nahverkehrs stellt die schiere Datenmenge dar. Diese wird vor allem dann zum Problem, wenn eine Echtzeitvisualisierung auf beliebigen Teilen des Datensatzes erfolgen soll (zum Beispiel, um eine interaktive Karte zu generieren). In diesem Fall liegt die Hauptaufgabe darin, die Menge an Fahrtverläufen auf die für die Visualisierung relevanten Teile zu reduzieren. In dieser Arbeit präsentieren wir einen Ansatz, der statische Fahrplandaten und dynamische Echtzeitinformationen zu Fahrtverläufen aufbereitet, die als stückweise lineare Funktionen verstanden werden und die Zeit auf Koordinaten der (projizierten) Kartenebene abbilden. Wir legen einen effizienten Server vor, der zeitliche und räumliche Grenzen als Eingabe akzeptiert und partielle Fahrtverläufe innerhalb dieser Grenzen ausgibt. Wir diskutieren die Details der Implementierung und zeigen, dass unser Ansatz problemlos in der Lage ist, mit den öffentlichen Nahverkehrsdaten ganzer Länder umzugehen. Darüber hinaus legen wir einen schlanken Vektor-Layer für Online-Kartendienste vor, der gleichzeitig tausende von Fahrzeugbewegungen (mit optionaler Zeitrafferfunktion) darstellen kann. Außerdem skizzieren wir einen Ansatz um für eine Liste von Stationspunkten den tatsächlichen Linienverlauf aus Geodaten zu extrahieren.

Acknowledgements

I would like to thank my supervisor Hannah Bast for giving me the opportunity to choose this topic, for proposing useful ideas and for providing excellent video lectures on programming and route planning. They have been my evening entertainment program for some weeks now.

Furthermore, my thanks go to Sabine Storandt for proofreading this thesis and to Christian Schindelhauer for agreeing to be secondary reviewer.

Special thanks go to Daniel Hofstetter of the Swiss Federal Railways (SBB) for giving me the opportunity to do tests on the complete Swiss railroad schedule of 2014 and to the entire team of geOps, Freiburg.

Finally, I want to thank my parents for supporting me throughout my unending years of study.

Contents

1. Introduction

Over the past years, there has been an increasing interest in visualizing public transportation networks, especially in displaying the live positions of vehicles. After the release of *swisstrains.ch*, a live map for the Swiss railway network in 2007, several transport agencies released their own tools for tracking live movements of trains, buses or streetcars. Notable examples include the train radars by Deutsche Bahn[1] and Österreichische Bundesbahnen[2] as well as the live timetable of Munich's S-Bahn-System[3]. Among others, unofficial projects exist for the Stuttgart region[4], the UK[5] and for the Freiburg area[6]. This list is far from complete.

Live maps of public transit networks are more than a mere toy for railroad enthusiasts. The increasing number of official maps leads to the assumption that there is a certain promotional value in visualizing the live network of a transportation agency. Other than with road networks, a static (possibly printed) map of a transit network is not able to show the capacity of the system. While roads are (in general) continuously accessible through fixed entry points, public transportation only allows for intermittent access in most of the cases. Restrictions include stop positions, vehicle types and, of course, the timetable. While it is easily possible to represent stations and lines on a map, it is difficult to give a complete static two-dimensional picture of the network's train coverage.

Live maps can help with that, but it is not their primary asset. The now discontinued train radar of the German newspaper "Süddeutsche Zeitung" used aggregated delay information to analyze the long-distance network of Deutsche Bahn for punctuality and compared it against official statistics. But visualization of real-time public transit data can not only be useful to inform passengers about delays. It can even help to straighten out minor shifts in the timetable by providing customers with actual positions of vehicles, enabling them to plan their journey accordingly. A possible use-case scenario could be a smartphone app that shows the position of every vehicle in the surrounding area along with their routes and the nearby stations.

[1] `http://bahn.de/zugradar`
[2] `http://zugradar.oebb.at`
[3] `http://s-bahn-muenchen.hafas.de`
[4] `http://www.nahverkehrskarte.de`
[5] `http://traintimes.org.uk/map`
[6] `http://tracker.patrickbrosi.de`

Their full potential can be unleashed by combining them with classic route planners. This would have gone beyond the scope of this thesis, but we discuss this idea as future work in Chapter 7.

Despite the various efforts put into the development of the services described above, there exists (at least to our knowledge) no universal and scalable toolset for the visualization of real-time public transit data. Existing approaches are either bound to the specific data representation of a single service agency or they have their own internal timetable format, very often scraped from various web services. All approaches that we know of are restricted either to a specific agency, a location (a region, a town, a country) or to certain types of vehicles (only trains, only buses, only streetcars, only airplanes).

This thesis aims to overcome these restrictions. We describe a fully scalable approach that uses static timetable data and real-time delay information to generate a live transit map. In Chapter 2, we name the challenges that lie in such a task and present basic approaches to master them. After that, Chapter 3 formalizes the problem of getting the current position of a vehicle out of a vast array of scheduled trips and real-time data. We proceed with a short introduction to map projections that enable us to treat vehicle trajectories as piecewise linear curves on the plane. In Chapter 5, we introduce TrajServ, a server that holds and reads General Transit Feed Specification data (for a more detailed introduction to GTFS, see Section 3.1.3), manages real-time data and outputs the cropped trajectories of vehicles inside a certain bounding box. We also present TRAVIC, a thin browser-based client that is able to display thousands of smooth vehicle movements on a map. We discuss the implementation details, explore certain speed-up techniques and evaluate the overall performance. We conclude with a list of possible future improvements, additional use cases and a description of ideas how to extract vehicle routes out of geospatial data using route planning or map matching.

To our best knowledge, this is not only the first implementation of a live transit map that is entirely based on GTFS, but also the first that is (given the required data) able to efficiently visualize the public transportation network of the whole world.

2. Challenges and Approaches

On a single Monday, there are 81,950 vehicle movements in the greater New York City area. Each of those vehicles (including buses) makes an average of 38 stops and each route is described by a polyline with an average of 284 vertices. On a normal Monday morning at 8:00 am, there are ∼4,700 vehicles in the networks of New York and New Jersey, including buses, trains, light rail and subways. Figure 2.1 gives an overview of the sizes of various networks around the world on a normal Monday. The data is taken from TrajServ and is calculated from the processed and optimized input GTFS.

Network area	trips	stations[1]	arr/dep events	vertices[2]	vehicles 8am
Netherlands	111,537	73,293	2,438,857	3,843,780	4,8 k
New York City[3]	81,950	34,948	3,126,850	3,482,713	4,7 k
Switzerland[4]	77,949	21,689	2,092,196	—[5]	2 k
Turin	16,399	7,250	498,524	198,946	820
Freiburg	7,595	1,611	97,535	—[5]	∼ 150
Vitoria-Gasteiz	1,766	338	35,867	4,198	65

Table 2.1: Parameters of several public transit networks on a single monday.

These numbers give an impression of the amount of data that has to be handled if we want to visualize the vehicle movements *of only a single metropolitan area on a single day*. A list of parameters for the whole datasets can be seen in Table 6.1. The primary challenge lies in efficiently passing these vast amounts of data through various bottlenecks to an end user's screen. These bottlenecks include the processing unit as well as the available memory, the interface between backend and frontend and the rendering engine on the client machine. We will see in Chapter 6 that naïve baseline approaches fail miserably in passing these bottlenecks and that additional optimizations are necessary.

[1] Total network number.
[2] Unique shape waypoints. Waypoints are shared between trips.
[3] MTA, LIRR, MNR, PATH, NJ TRANSIT.
[4] SBB, BLS, RhB, BERNMOBIL, Transports publics de la région lausannoise, Baselland Transport and others.
[5] No line shape information available.

Without transit data, of course, the bottlenecks will never clog and optimizations would be like fighting windmills. We need data sources. In this chapter, we therefore begin with discussing the advantages and disadvantages of static schedules and real-time data. We explain why we think that neither of them are useful on their own for a live transit map and introduce a combined approach. After that, we take a look at possible client/server interfaces, describe the two general approaches that are in use today and explain which alternative was chosen for TrajServ.

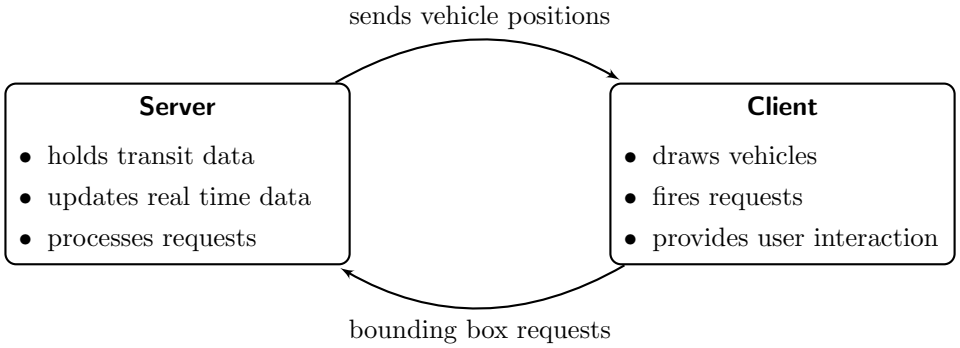

Figure 2.1: Architecture of a live transit map.

From now on, we will focus on a client/server architecture that resembles the one depicted in Figure 2.1. The *server* manages transit data, receives requests and outputs information that enables the *client* to display vehicles on the screen. A client can either be a web application, a desktop application, a smartphone app or something completely different. We try to be as implementation-independent as possible during the next chapters.

2.1. Static vs. Real-Time Data

Static schedule data describes the state a transportation network is *supposed* to be in at an arbitrary time t. Real-time transit data describes the state a transportation network is *currently* (t_{cur}) in. As avid train users know, there can be a huge difference between realtimeState(t_{cur}) and staticState(t_{cur}). Hence, settling for a transit visualization that is based on static schedule data can lead to results that have little to do with reality. Real-time data, on the other hand, is rarely available and almost never network-complete.

2.1.1. Raw GPS Data

There are multiple ways to describe the current real-time state of a transit network. The most intuitive, however, is a simple list of all vehicles currently moving through

the network, together with their actual GPS position. The positions can be aggregated, for example, by GPS devices on every vehicle. A visualization based on data like this would always show the state a network is *really* in. However, there are multiple reasons why it is generally a bad idea to rely on raw GPS data alone, at least in the year 2014 (and probably for many years to come.)

Low availability and coverage There are extremely few transportation agencies that really provide raw GPS positions of their vehicles. In fact, we don't know of a single one. Even if an agency has GPS trackers on a few of their vehicles, there are often smaller lines without tracking devices. Additionally, temporal coverage is usually very bad, with vehicles sending their current position only every few seconds.

No fall-back Relying on raw GPS positions means that if a tracking device fails, the visualized vehicle will stop or even disappear. There is no fall-back.

No extrapolation Raw GPS positions are completely semantic. The data does not reveal the underlying structure of the transit network at all. A typical server output looks like this: vehicle a of line t is currently at position (x, y). We generally don't know the route of the vehicle, the remaining time until it arrives at the next station or the vehicle's process on the current part of its route. Therefore, it is impossible to predict the way a vehicle will take during the next few minutes. We can't do lookahead requests because we cannot even tell where the vehicle will be in two seconds. It is possible to calculate this information by means of matching the position and line number against a complete map of the network, but to do that we would need static data and enter the realms of the combined approach (see Section 2.1.3).

Amount of data Because the client is not able to cache, we have to update the GPS positions periodically. This means heavy traffic on the client/server interface. We discuss this problem more detailed in Section 2.2.1.

2.1.2. Interpolated Schedules

A robust method to visualize vehicle movements is to do an interpolation of the official static schedule. We describe the formal details to this approach in Section 3.1.4. It has none of the disadvantages described in the section above. Static schedules can be cached on the client side, they allow for visualization of the network's (scheduled) state at arbitrary timepoints with arbitrary time-lapsing and they are much more easy to obtain than GPS positions. However, the obvious disadvantage of this approach is the lack of real-time information. If a vehicle is delayed, canceled or has to change its route because of an accident or constructions, the approach is not able to display this.

2.1.3. Combined Approach

In Section 2.1.1 we mentioned the impossibility of doing a correct spatial or temporal extrapolation on raw GPS data. What we did not mention is that most modern transit agencies have to solve this problem all the time. When you are standing at a streetcar stop with a display that shows the minutes remaining until the next streetcar, you see a temporal interpolation of the arrival time based on the vehicle's current position[6]. As hinted in Section 2.1.1, this is done by comparing the vehicle position to the schedule. The difference between the extrapolated arrival time at station A and the scheduled one is the delay δ. Knowledge of the current delay δ of a vehicle allows us to do a spatial interpolation that is nearly equivalent to the real-word position. In general, the smaller the distance between control points, the smaller the deviation. Note that even in the worst case scenario where only stop points are control points, the deviation will only be noticeable en route and converges to 0 when the vehicle approaches the station. Many transportation agencies output the delay information of their vehicle.

We call the extension of static schedules with live delay information for certain control points the combined approach. It has all the advantages of static schedule interpolation (robustness, cacheable look-ahead requests, time-lapsing) and is nearly congruent with real-world positions. This approach adds minimal additional information to each timepoint (the delay δ) and will fall back seamlessly to the static schedule if no real-time information is available. TrajServ uses the combined approach the calculate vehicle positions.

2.2. Client-Server-Interface

Figure 2.1 shows that the interface between client and server is a crucial bottleneck of the visualization. In this section, we present the two most common interface designs currently used in transit visualization maps. They closely correspond with the data formats described in the previous section.

2.2.1. Periodical Updates

The basic method of displaying vehicle movements on a map is to fire periodic position requests for a certain area of the network and to draw the results on the map. If client performance is low, this can be the preferred way to visualize vehicles,

[6] In many cases, the interpolation is not based on the actual GPS coordinate of the vehicle, but on certain control points that are positioned along the routes. Here, the static spatial network data is "built into" the system. If a vehicle passes control point A, the system knows the time the vehicle should be at A according to the schedule and is able to calculate a delay.

as the client code can be extremely thin. In fact, for web applications, most map APIs like Google Maps, OpenLayers or Leaflet provide methods that make it possible to get a working client with only a few lines of code. Many of the transit visualization maps described in Chapter 1 use this approach.

If the server uses raw GPS positions as a data source, this is the only practicable interface. Nevertheless, it is also possible to use this approach with interpolated schedule data.

There are several flaws in this method. Its biggest advantage, the absence of any client code to calculate vehicle positions, is also its biggest disadvantage. Without a new position request, vehicles won't move. To achieve a smooth simulation, the requests have to be repeated frequently. This generates a lot of server load. If client and server are physically separated, it also means heavy network traffic. This is especially problematic on mobile devices, where network connections can be extremely slow or even break down completely. If the server connection is interrupted, the vehicle movement stops. This problem is related to the missing GPS updates problem described in Section 2.1.1.

2.2.2. Spatiotemporal Queries

A better interface design that leverages the combined approach's ability to do look-ahead requests are *spatiotemporal queries*. In this design, the client does not request vehicle positions, but partial vehicle trajectories within a certain rectangle for a specific timespan Δt (e.g. 2 minutes). We call this request parameter a *spatiotemporal bounding box*.

Definition 1. *A spatiotemporal bounding box B_{st} is a 6-tuple $(x_1, y_1, x_2, y_2, t_b, t_e)$ where x_1, y_1 is the lower left (we say: south-west) corner of a rectangle and x_2, y_2 the upper right one (we say: north-east). B_{st} is additionally bounded by begin time t_b and end time t_e.*

Note that we only operate in two-dimensional space. We can now formally define a spatiotemporal query interface.

Definition 2. *A spatiotemporal query interface requires the client to request trajectories in $B_{st} = (x_1, y_1, x_2, y_2, t_b, t_e)$ with frequency $f = (\Delta t - \theta)^{-1}$ to achieve complete temporal coverage, where $\Delta t = t_e - t_b$ and θ is the overall interface delay (network latency, server processing time etc.)*

The client has to make another request only after exceeding B_{st}. This means that as long as the client stays within the rectangle (x_1, y_1, x_2, y_2), it only has to do requests every $\Delta t - \theta$ time units. If B_{st} is spatially padded, it is even possible to

navigate through a certain area without the need for new requests. Note that vehicle delays are only transmitted on each new request, which means that the client tends to be inaccurate for small f. We describe this problem in Section 6.3.

With the definition of a spatiotemporal interface, it is now easy to see that periodical update queries are in fact a special case of spatiotemporal queries.

Corollary 1. *A periodical position-update interface is a special case of a spatiotemporal query interface where* $t_b = t_e$.

The frequency has then to be $f = (-\delta)^{-1}$ for complete temporal coverage. Even in the best case scenario where $\delta = 0$, the update frequency would have to be infinitely large. If we assume the screen refresh rate to be 60 Hz, δ would have to be ≤ 16 ms to get a smooth visualization, which is unrealistic if you consider communication via HTTP requests and a server that has to answer 60 requests per second from possibly hundreds of clients. The bottleneck will definitely clog.

This is the reason why it is generally not possible to get smooth vehicle movements with periodical updates and spatiotemporal queries are preferable.

3. Data Structures

The primary application of public transit schedules is route planning. However, using transit data for a live vehicle movement visualization is essentially different than using it for routing purposes. In route planning, the goal is to find shortest paths through the transportation network very fast, beginning at a certain entry point. If we want to display vehicle movements, we have to efficiently get *all* (partial) paths of *all* vehicles passing through a certain spatiotemporal bounding box. Basically, this means that data structures that were designed for route planning are not suited for movement visualization per se. In this chapter, we formally describe data structures for the combined approach that allow for fast processing of spatiotemporal queries. The basic data structure of our model is a vehicle trajectory $\mathcal{T}$ which can be understood as the complete spatial and temporal path of a *single* vehicle. We use the formal term *trajectory* and the GTFS term *trip* somewhat interchangeably, but there are subtle differences between them.

3.1. Modeling Static Trajectories

Two data structures that are commonly used in route planning to model schedule data are time-expanded and time-dependent graphs [1]. In this section, we explain why they are not useful when it comes to live maps and describe an explicit approach to model static transit data.

3.1.1. Implicit Approach

Both the time-expanded and the time-dependent models transform static schedule data into a directed graph $G = (V, E)$. In the time-dependent model, each $v \in V$ models a station and each $e = (u, v) \in E; u, v \in V$ models possible non-stop connections between two nodes. This is an intuitive approach. The schedule is modeled by introducing special cost functions $c_{u,v}(t)$ that respect the travel time as well as the waiting time until the next departure. If, for example, there is a connection from u to v that departs at 12pm and 2pm and takes 2 hours, $c_{u,v}(10) = 4$, $c_{u,v}(12) = 2$, $c_{u,v}(13) = 3$ and so on. For a detailed explanation of a shortest-path-algorithm (e.g. Dijkstra) that works on time-dependent graphs, see [2]. Along with

departure times, line names and other vehicle attributes can be stored inside the connection edges. It is possible to extend this model with a second class of nodes that represent vertices of a polyline to store the exact trajectory between two stations.

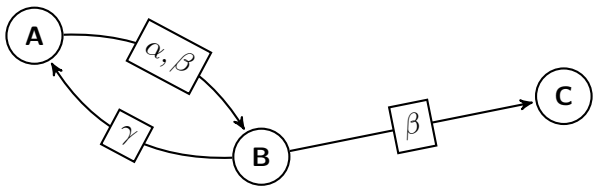

Figure 3.1: Time-dependent graph with 3 stations A, B and C connected by vehicles α, β and γ.

In terms of modeling a transit network for a live movement visualization, time-dependent graphs can be very space-efficient. They allow sharing route information as well as station information, because trips are modeled implicitly on a shared network of stations and routes. But it comes at a cost. Consider a basic request like *"output the complete trajectory of trip $\mathcal{T}$ on day d"* on a time-dependent graph. First, we would have to scan the set of nodes for the start node v_1 of $\mathcal{T}$. Then we would have to follow an edge that represents the connection $\mathcal{T}$ provides on d to the next station node on its route, v_2. In v_2, we would again have to check which edge belongs to $\mathcal{T}$ and follow it to v_3 and so on. We would have to execute a basic routing algorithm, simply because $\mathcal{T}$ is not stored explicitly as a single entity. Now consider a request that asks for all vehicle positions in a rectangle R at time t. First, we look up the set S of all station nodes within R. For each $v \in S$, we now have to check every outgoing edge (v, u) for vehicles that are currently between v and u. Even in a small network like Freiburg's with ∼1,600 station nodes, the number of edges can be very high. The problem gets even more complex if you consider a spatiotemporal request. For each trip that is implicitly described in S we have the check whether it traverses the spatiotemporal bounding box B. We can't just omit certain $v \in S$, because each v possibly stores thousands of different departures and we don't know whether one of these departures lies within B. In the end, even for small networks, we have to check millions of non-stop connections between station nodes just to get the positions of only a handful of vehicles currently within B.

This exemplifies that time-dependent graphs are not a very useful data model when it comes to visualization of vehicle movements. One of the reasons for their impracticalness is that it is very difficult to access vehicle trajectories that are relevant in the current timespan. Everything is connected with everything else in some way or another. While this is essential for route planning, we don't need transfer connections between trajectories at all for movement visualization.

3.1.2. Explicit Approach

In the time-expanded approach, departures and arrivals are modeled as own nodes in the graph [1]. Arcs between a departure and an arrival node are non-stop trips of a single vehicle. To model changes of trains, the time-expanded model uses transfer nodes. One might argue that in the time-expanded approach, trajectories are in fact modeled explicitly because each arrival and each departure node belongs to a single trip. While this is partly true, time-expanded graphs still model information that is not needed for a live visualization (transfer nodes and waiting edges). Additionally, the explicit storage of arrivals and departures as single nodes is not space-efficient.

In the next two sections, we describe how trajectories are represented in the GTFS format and formalize trajectories as piecewise linear curves.

3.1.3. GTFS

During the last years, the General Transit Feed Specification has become the most popular format to describe static schedule data of transit networks [7]. Both official or user-generated feeds are available for a huge number of transit agencies around the world [8]. GTFS can model weekly schedules as well as explicit day-wise trips, provides the possibility to store polylines ("shapes") for single or multiple trips, is able to integrate multiple trips into a single service that is presented to network users ("route") and can hold many other attributes like wheelchair-accessibility, route colors or fare information [3].

A GTFS feed consists of 13 CSV-files (some of them optional). For a better understanding of Chapter 5, we give a short overview of the ones that are most important for the purposes of this thesis.

agency.txt Holds information about one or multiple service agencies of this feed. This file also holds the timezone. This is important because times are always provided in a HH:MM:SS format, never as absolute timestamps.

stops.txt A list of all stations along with their respective IDs, human-readable names and geographical positions.

trips.txt The headers of all vehicle movements in this feed. Each trip has a service ID which specifies the days it operates on.

stop_times.txt The exact station sequence for each trip. Each station has an arrival and a departure time.

calendar.txt Holds weekly service times referenced by trips.txt.

calendar_dates.txt Holds explicit date-wise service times referenced by `trips.txt`. These services can extend services specified in `calendar.txt` with single *exceptions*, but they can also stand alone.

shapes.txt Representation of geographical polylines that describe the exact route a vehicle takes.

routes.txt Groups trips into single services that are presented to users.

A complete UML diagram describing the relationships and their cardinalities can be found at [5]. For now, please note that despite the fact that vehicle trips are stored explicitly, regular services only have to be specified a single time. Their repetitive nature can be declared in `calendar.txt`. Also note that it is possible for geographical polylines in `shapes.txt` to be shared between trips. This also holds for services described in `calendar.txt` and `calendar_dates.txt`.

3.1.4. Vehicle Trajectories

Based on the GTFS format, we can now formally define vehicle trajectories as explicit entities. We describe a trajectory as a list of spatiotemporal waypoints. These points can either be stations or vertices of a polyline. A trajectory describes the way a single vehicle takes through time and two-dimensional space.

Definition 3. *A spatiotemporal waypoint p is a 3-tuple (x, y, t) where x, y are coordinates on the two-dimensional spatial plane and t is a timestamp. The set of all p of a trajectory is $\mathcal{P}$.*

More formally, we can say that all $p \in \mathcal{P}$ form a parametrization (with parameter t) of a piecewise linear curve in $\mathbb{R}^2$.

Definition 4. *A two dimensional piecewise linear curve is a curve $C : [0, n] \mapsto \mathbb{R}^2$ where $n \in \mathbb{N}$ and $\forall i \in \{0, 1, ..., n-1\}$: the part of C between i and $i+1$ is linear. A parametrization of C is a mapping $q : Q \mapsto [0, n]$.*

In practice, we combine arrival and departure points p_a and p_d into a single $p_{station}$ that holds both the arrival and departure times t_a and t_d.

Note that in GTFS, t is not an absolute timestamp but a time relative to the current service day. A vehicle defined by a trajectory can only appear once a day. We say a trajectory is *active* for a specific date d if its activity function evaluates to 1 for d.

Definition 5. *An activity function α is a function $\mathcal{D} \mapsto (0, 1)$ that is described as follows:*

$$\alpha(d) = \begin{cases} 1 & \text{if there is a service on } d \\ 0 & \text{if there is no service on } d, \end{cases}$$

where $d \in \mathcal{D}$ and $\mathcal{D}$ is the set of valid dates.

With this, we can formally describe trajectories per service day.

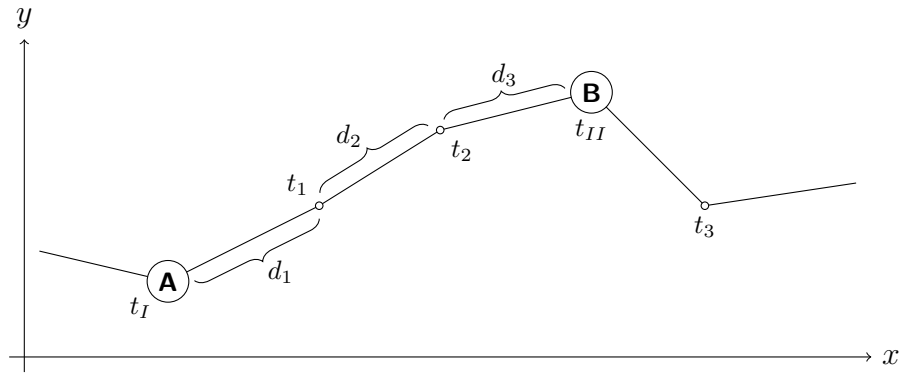

Figure 3.2: Temporal interpolation of non-timestamped waypoints between stations A and B.

Definition 6. *A trajectory $\mathcal{T}$ is a 2-tuple $(\mathcal{P}, \alpha)$ where α is the activity function and $\mathcal{P} = \{p_1, ..., p_n\}$, $p_i = (x_i, y_i, t_i)$ the set of spatiotemporal waypoints. $\forall i, j \leq n : i \leq j \Rightarrow t_i \leq t_j$.*

As mentioned above, $\mathcal{P}$ describes a vehicle's route as a piecewise linear curve in $\mathbb{R}^2$. This allows for fast clipping and interpolation algorithms. We point out that, despite the obvious computational advantage, it is not necessary to model a vehicle's spatial route as a piecewise linear curve. We could also see the spatial components of all p_i as knots of a piecewise polynomial curve to achieve smoother interpolation results. However, we argue that such an approach would be out of proportion here.

In Chapter 5 we frequently use the concept of a trajectory's minimum spatiotemporal bounding box.

Corollary 2. *Each trajectory $\mathcal{T}$ has a minimum spatiotemporal bounding box $B_{min}(\mathcal{T})$ that describes the temporal and spatial extent of $\mathcal{T}$.*

We point out that there is a subtle difficulty with Definitions 6 and 3. In real-world transit schedules, we usually don't have a timestamp for every p. Temporal information is only provided for station waypoints, with distance-ordered waypoints in between. To get a trajectory $\mathcal{T}$ as defined in Definition 6, a temporal interpolation is necessary.

Figure 3.2 shows the general approach to get timestamps for all $p \in \mathcal{P}$. We assume a constant speed between A and B. With

$$\begin{aligned} d_{AB} &= \text{dist}(A, B) = d_1 + d_2 + d_3 \\ t_{AB} &= t_{II} - t_I \end{aligned}$$

we get the interpolated timestamps

$$\begin{aligned} t_1 &= \left(\frac{d_1}{d_{AB}} \times t_{AB}\right) + t_I \\ t_2 &= \left(\frac{d_1 + d_2}{d_{AB}} \times t_{AB}\right) + t_I \\ t_3 &= \left(\frac{d_1 + d_2 + d_3}{d_{AB}} \times t_{AB}\right) + t_I. \end{aligned}$$

This can be calculated either during construction of $\mathcal{T}$ or *on-the-fly* during execution of the clipping and interpolation algorithms. We will discuss the advantages of both approaches in Chapter 5.

After doing the temporal interpolation, given a time t_{cur} between two waypoints p_i and p_{i+1} with timestamps t_i and t_{i+1} ($t_i \leq t_{cur} \leq t_{i+1}$), their respective delays δ_i, δ_{i+1} and spatial coordinates $(x_i, y_i), (x_{i+1}, y_{i+1})$, the current vehicle position can be calculated using the equations

$$\begin{aligned} x_{\mathcal{T}}(t_{cur}) &= \left[\frac{(x_{i+1} - x_i)\,(t_{cur} - (t_i + \delta_i))}{(t_{i+1} + \delta_{i+1}) - (t_i + \delta_i)}\right] + x_i, \\ y_{\mathcal{T}}(t_{cur}) &= \left[\frac{(y_{i+1} - y_i)\,(t_{cur} - (t_i + \delta_i))}{(t_{i+1} + \delta_{i+1}) - (t_i + \delta_i)}\right] + y_i. \end{aligned}$$

To allow for continuous spatial and temporal clipping of $\mathcal{T}$, we introduce a partial trajectory.

Definition 7. *Given a trajectory $\mathcal{T} = (\mathcal{P}, \alpha)$, we call $\mathcal{T}^{par} = (\mathcal{P}^{par}, p_b, p_e, \alpha)$ a partial trajectory of $\mathcal{T}$. $\mathcal{P}^{par} \subseteq \mathcal{P}$.*

In other words, we add new starting and end points p_b and p_e that mark the exact positions where a spatiotemporal bounding box B clips $\mathcal{T}$. If multiple parts of $\mathcal{T}$ lie within B, we use multiple $\mathcal{T}^{par}$.

Definition 8. *If a partial trajectory $\mathcal{T}^{par}$ belongs to a trajectory $\mathcal{T}$, we write $\mathcal{T}^{par} \subseteq \mathcal{T}$. We say $\mathcal{T}^{par} = \mathcal{T}$ if both have the same activity function α, $\mathcal{P} = \mathcal{P}^{par}$, p_b is equal to the first waypoint in $\mathcal{P}$ and p_e is equal to the last waypoint in $\mathcal{P}$.*

3.2. Multi-Layer Grids

In Chapter 2, we gave the sizes of several transit datasets. A naïve approach to manage the set of vehicle trajectories inside a server would be a simple list. However, for bigger datasets like the transit schedules of entire countries, such an approach is

very inefficient. For each spatiotemporal request, *each* trajectory (possibly millions, see Figure 2.1) has to be checked for parts that are inside the bounding box. This becomes especially ineffective if the servers holds data from around the globe. Why should we check for vehicles that currently move around in Boston when in fact we are only looking at the city center of Turin?

A common index structure for geo-coordinates is an R-tree [9]. R-trees group nearby objects and represent them with their minimum bounding rectangle. Multiple bounding rectangles can again be grouped by their respective bounding rectangles on the next level. R-Trees allow for fast nearest-neighbor and bounding rectangle requests, and they are commonly used in geographic information systems.

However, as we will see in Chapter 5, we usually want to create a hierarchy among trajectories. City buses, for example, should only appear in the live transit map if we are above a certain zoom level. Subways or streetcars should not be visible if we are at a very low zoom level and looking at an entire country. While this could be modeled by using multiple R-trees for each zoom level, we still would have to traverse each level's R-tree if we wanted to output the vehicles on multiple levels. Trains, for example, should appear on the highest zoom level along with buses, streetcars or ferries.

To allow for hierarchical indexing, we chose the approach of a classic index grid. A grid contains $N \times N$ grid cells with sizes $n \times n$. It can be stored in a simple two-dimensional array. To model the hierarchy, we use a multi-layer grid. Like an R-tree, it groups multiple bounding rectangles into a bigger rectangle on the next level. Figure 3.3 gives an example of such a multi-layer grid. Each layer is visible on a certain zoom level. The side lengths of a cell on level n are two times the lengths of a cell on level $n + 1$. This resembles the way tiles are generated for web map services like OpenStreetMap or Google Maps. It can be seen as a tree with a single cell of side lengths l_r as a root and each vertex with side length l having exactly four child nodes with side lengths $\frac{l}{2}$. In Figure 3.3, a rectangle request for zoom level 15 is highlighted along with the grid cells that have to be checked for trajectories traversing the rectangle. The obvious disadvantage of this approach is that we usually have to index a trajectory in multiple cells. However, this can be minimized by choosing appropriate side lengths.

To index the temporal component of a trajectory, we chose a discrete approach that sorts trajectories into multiple date bins, based on their activity function. In TrajServ, we use 9 bins per grid cell. Bins 1-7 model weekdays. For example, a trajectory is indexed in bin 2 if it is active on Tuesdays. Services that are active on workdays are stored in bin 8. Bin 9 is reserved for irregular trajectories that are only active on specific dates. Those bins can be implemented using a single array in which trajectories are sorted by their bin number and by storing pointers to the first element in each bin. Trajectories in bin 9 are additionally sorted by date.

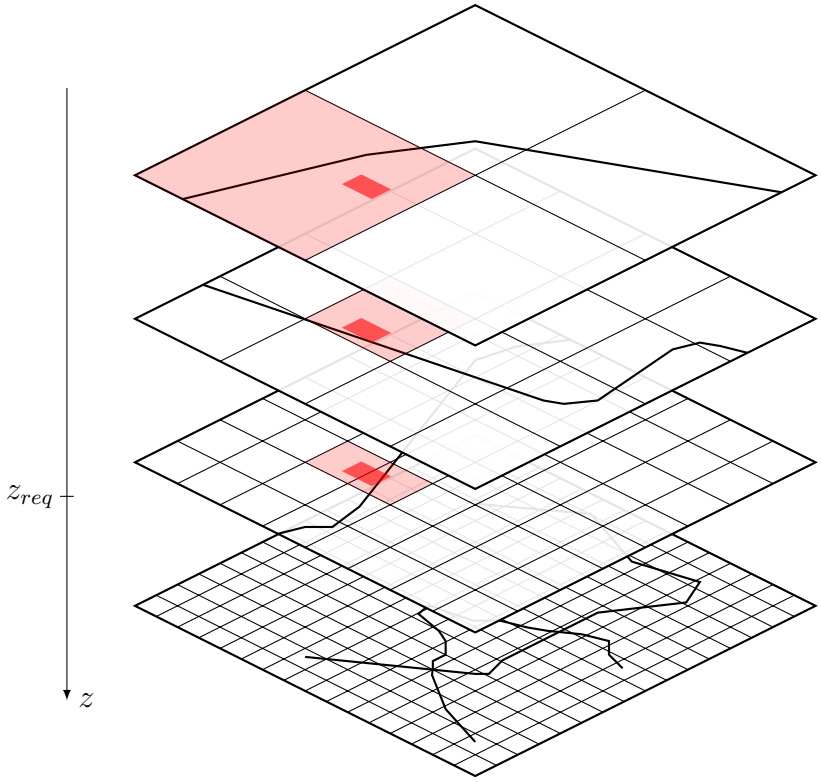

Figure 3.3: Example of a multi-layer grid. The current view-box and the grid cells that have to be checked are highlighted.

3.3. Modeling Real-Time Data

As mentioned earlier, real-time public transit data is usually modeled as a list of arrival and departure delays per trip and station.

Since August 2011, the GTFS specification is extended by a real-time transit data feed. GTFS-realtime provides support for three types of information [4].

Trip updates describe deviations from the official schedule like delays ("stop time updates"), cancellations and route changes. This is the main source of real-time data for TrajServ. Updates are given with the IDs of the scheduled GTFS trip they relate to.

Service alerts are general textual passenger announcements valid for a certain time span.

Vehicle positions is a feed for GPS vehicle positions. This can be used for an approach like the one described in Section 2.2.1, where actual vehicle coordinates are outputted explicitly, along with their current speed. At the time of writing this thesis, this feed is usually not provided.

4. Map Projections

In Section 3.1.4, we treated the spatial part of vehicle trajectories as curves on the two-dimensional plane. However, public transit vehicles move around on the earth surface, which is not flat. In this section, we briefly discuss several pitfalls that are related to the ellipsoidal nature of the earth.

4.1. Plane vs. Ellipsoid

Figure 3.2 gave an example of a trajectory's spatial path (a polyline) and the temporal interpolation of non-timestamped waypoints. We implicitly assumed that the shortest path between two vertices V_1 and V_2 is $\overline{V_1V_2}$. This is not entirely correct. In GTFS, vertices of vehicle trajectories are given as WGS 84 latitude and longitude coordinates [3, 12]. The shortest path between two points on the earth surface is not a straight-line segment, but part of a great-circle (an orthodrome) [13] between V_1 and V_2 (Figure 4.1). This means we cannot, in general, treat vehicle polylines consisting of WGS 84 vertices as piecewise linear functions. Both the spatial and temporal interpolations would be wrong. Note that this only becomes noticeable for large distances between coordinates. For smaller distances, the shortest path converges to a straight line.

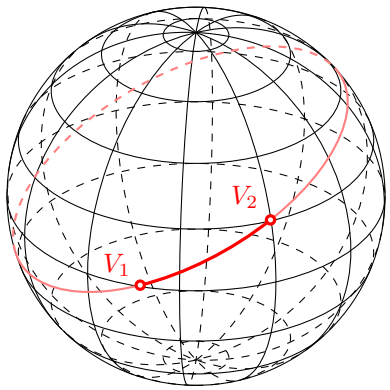

Figure 4.1: Great circle between two geo-coordinates.

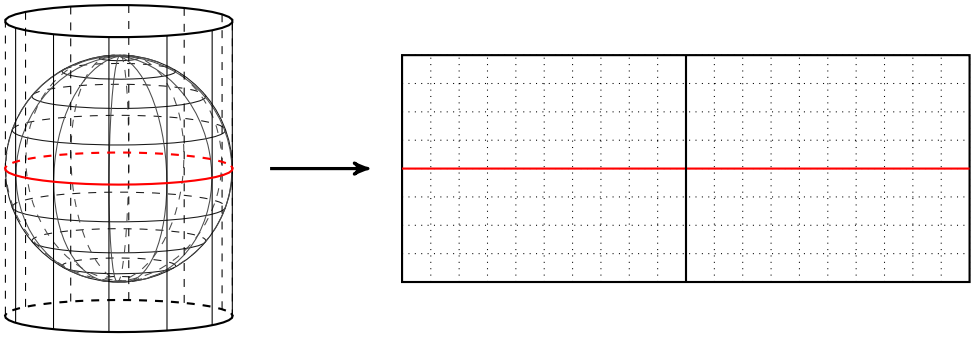

Figure 4.2: A cylindrical projection.

A complete introduction to spherical geometry would go beyond the scope of this thesis. The computational cost that comes with it is best illustrated by comparing the distance formulas. While on the two-dimensional plane, the Euclidean distance formula $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ is comparatively easy to compute (two subtractions, two multiplications, one addition, one square root), on the sphere the distance formula becomes

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right),$$

where ϕ_1 and ϕ_2 are latitudes, λ_1 and λ_2 are longitudes of two geo-coordinates [14]. We point out that a single trajectory consists of hundreds of vertices and that to answer a single request, the server usually has to do spatiotemporal interpolations of thousands of trajectories.

For completeness, we hint that even a great-circle is only an approximation of the shortest path between two geo-coordinates because the earth is not a perfect sphere [11].

4.2. Mercator Projection

One possibility to get back into the known waters of Euclidean geometry is to use a map projection. Map projections transform latitudes and longitudes of points on a sphere (the earth) into locations on a plane [11]. Figure 4.2 shows the basic principle of a cylindrical projection. A classic cylindrical projection is the Mercator projection. It was originally developed for navigational purposes but is today widely used for map services (EPSG:3857). The spherical Mercator projection approximates the earth surface with a sphere, not an ellipsoid.

A projection usually consists of two functions $x(\lambda)$, $y(\phi)$ which map a longitude λ and a latitude ϕ to their Cartesian coordinates on the plane. For the spherical Mercator projection, their most basic form is

$$x(\lambda) = \lambda - \lambda_0 \tag{4.1}$$

$$y(\phi) = \ln\left[\tan\left(\frac{\pi}{4} + \frac{\phi}{2}\right)\right], \tag{4.2}$$

where λ_0 is the central meridian (usually zero [11]). The functions map λ and ϕ onto a plane of width 2π. Since $y(\phi)$ becomes infinitely large at the poles ($\phi = \pm\frac{\pi}{2}$), it has to be truncated somewhere below 90°. Common street map services truncate at $\pm\hat{\phi}$, where $\hat{\phi} = 2\arctan(e^{\pi}) - \frac{\pi}{2}$ ($\approx 85.0511°$). Because $y(\hat{\phi}) = \pi$, this leads to a perfectly quadratic map.

To display x and y coordinates on the screen, we have to transform them into pixel coordinates. Most services use *tiles* as a basic unit for the size of their maps. A tile usually consists of 256×256 pixels [6]. On the lowest zoom level ($z = 0$), the map fits into a single tile. As mentioned in Section 3.2, the side lengths of tiles double with each zoom level, resulting in a total map side length of 256×2^z for each zoom level.

The origin of the standard Mercator projection is the intersection point of the null meridian and the equator. TrajServ uses a projection where the origin is on the lower left corner of the map, which means we have to shift coordinates. With this, we get the pixel projection functions

$$\begin{aligned} x_p(\lambda) &= \left(\frac{x(\lambda)}{2\pi} + \frac{1}{2}\right) \times l_t \times 2^{z_{max}} \\ &= \left(\frac{\lambda - \lambda_0}{2\pi} + \frac{1}{2}\right) \times l_t \times 2^{z_{max}} \end{aligned} \tag{4.3}$$

$$\begin{aligned} y_p(\phi) &= \left(\frac{y(\phi)}{2\pi} + \frac{1}{2}\right) \times l_t \times 2^{z_{max}} \\ &= \left(\frac{\ln\left[\tan\left(\frac{\pi}{4} + \frac{\phi}{2}\right)\right]}{2\pi} + \frac{1}{2}\right) \times l_t \times 2^{z_{max}}, \end{aligned} \tag{4.4}$$

where l_t is the tile side length and z_{max} is the biggest zoom level the server should be able to handle. TrajServer handles requests up to $z = 20$ by default, which results in a total projected map size of 268,435,456 × 268,435,456 pixels. Since coordinates are both stored as unsigned 32 bit integers, the maximum possible zoom level is $z = 32$.

The inverse projection functions are

$$\lambda(x_p) = \left(\frac{x_p}{l_t \times 2^{z_{max}}} - \frac{1}{2}\right) \times 2\pi + \lambda_0 \tag{4.5}$$

$$\phi(y_p) = 2 \arctan\left[\exp\left(\left(\frac{y_p}{l_t \times 2^{z_{max}}} - \frac{1}{2}\right) \times 2\pi\right)\right] - \frac{\pi}{2}. \tag{4.6}$$

After applying (4.3) and (4.4), we operate on a two dimensional plane and can safely handle trajectories as piecewise linear functions. It is important to note that even on the projected plane, the (real) shortest path between two vertices is not always a straight line, but can still be a curve. However, we argue that vertices of vehicle trajectories are very close together (usually the distance is below 100 meters), which means we can safely approximate the path the vehicle will take as a straight line. In addition, polylines of trajectories are usually represented with a planar map in mind. Large distances between anchor points (> 500 km) usually mean that the trajectory is only a rough approximation. In this case, straight lines seem more intuitively correct then bent curves.

We close with two another advantages the spherical Mercator projection has besides the safe theoretical ground. After applying equations (4.3) and (4.4), the server only has to operate with 32 bit integers. Except for a few remaining cases where high precision is necessary (for example for distance calculation during spatial interpolation), there is no need for floating point operations anymore. Another big advantage is discussed in Section 5.2. If the server already stores and outputs projected coordinates, the client is relieved from the task of doing the projection itself.

5. Implementation

In the previous chapters, we discussed the challenges that lie in creating a live transit map, described and evaluated possible approaches, defined models for storing and indexing vehicle trajectories and briefly explained map projections. In this chapter, we combine these ideas and concepts to develop a scalable, efficient real-time public transit map. We introduce TrajServ, a server that is able the answer spatiotemporal vehicle requests very fast and can, in theory, be used with arbitrary clients. After that, we introduce TRAVIC, a lightweight web-client that uses TrajServ data to draw a live transit vector map layer. The general architecture of TrajServ and TRAVIC has been shown in Figure 2.1.

5.1. TrajServ

TrajServ is a server written in C++ that holds transit data in a layered grid structure (Section 3.2), requests real-time data from feeds and responds to spatiotemporal requests (Section 2.2.2) sent by the client via a HTTP-interface. TrajServ's architecture and workflow are depicted in Figure 5.1. Listing 5.1 shows a partial JSON answer outputted by TrajServ. Note that in the `p` fields, coordinates are not given as latitudes and longitudes but as projected integer pixel coordinates relative to the lower left corner of the client's viewbox.

In this section, we highlight some difficulties related to the loading of GTFS files and discuss the construction of the grid layer. We describe the central cropping and interpolation algorithm and illustrate several techniques that are used to speed up request processing. Section 5.1.4 covers the usage of real-time data and delay updates.

5.1.1. Data Loading

In Section 3.1.4, we spoke of a trajectory $\mathcal{T}$ as a single entity that combines stop information and line shape information. The GTFS format, however, separates them [3]. This makes sense because line shape information is not necessarily part of a schedule. Additionally, in most cases different vehicle trips share the same line shape (e.g. regular services). Internally, TrajServ also tries to share line shape

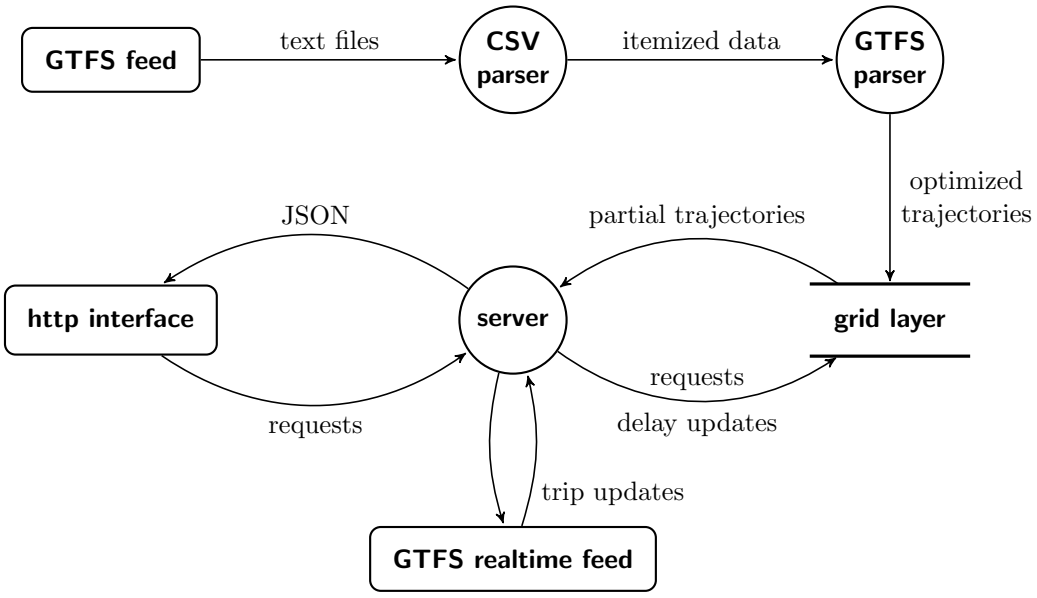

Figure 5.1: General architecture and workflow of TrajServ.

information between trips. However, at least at the current state of development, this sharing is optimized for access time, not for space.

For spatial interpolation, TrajServ needs to associate the stop time information stored in `stop_times.txt` (Listing 5.2) and the waypoint information stored in `shapes.txt` (Listing 5.3). GTFS provides an easy way to do this by means of the `shape_dist_traveled` field. If `shape_dist_traveled` is available, each waypoint in `shapes.txt` stores a distance that has been traveled if the vehicle reaches this waypoint. The unit of `shape_dist_traveled` is arbitrary and can be in miles, kilometers, meters, percentage of the total route or something completely different. Each line in `stop_times.txt` also provides a distance in the same unit that has been traveled if the vehicle arrives at a stop. Stop waypoints and shape waypoints can easily be combined by sorting them by `shape_dist_traveled`.

However, `shape_dist_traveled` is an optional field. Very few GTFS feeds actually provide it. One of the main reasons (besides lack of ambition) is the fact that it bloats the data. The GTFS standard dictates that both in `stop_times.txt` and

```
T1,8:54:00,8:55:00,S6,11
T1,8:59:00,8:59:00,S7,12
T1,9:05:00,9:07:00,S8,13
```

Listing 5.2: stop_times.txt

```
Shp4,47.991252,7.854387,71
Shp4,47.991403,7.85439,72
Shp4,47.991636,7.854377,73
```

Listing 5.3: shapes.txt

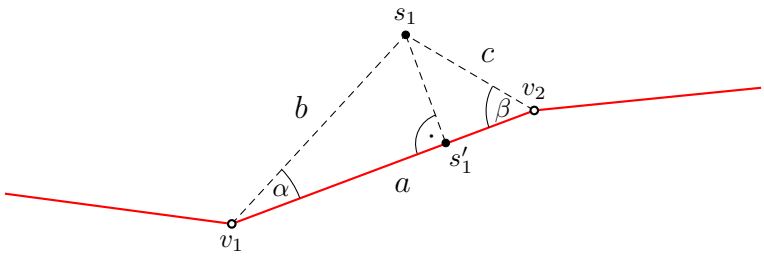

Figure 5.2: Point-to-Point matching with two shape vertices (v_1 and v_2) and a station (s_1).

`shapes.txt`, `shape_dist_traveled` has to increase along the route. This means that trips of the same route can only share line shape information in the same direction. For reverse travel, another shape has to be stored that is essentially the same, but backwards. Since most routes are served in both directions, this usually means that the size of `shapes.txt` doubles if `shape_dist_traveled` is introduced.

TrajServ does use `shape_dist_traveled` if present, but it does not rely on it. One of the first things the GTFS-Parser does after reading the feed and projecting the coordinates onto the plane is to merge shape waypoints and station waypoints into a trajectory $\mathcal{T}$. This is done by an algorithm that respects the sequence of both the shape and the station waypoints as well as the order of potential `shape_dist_traveled` fields. If there is no such field present for all or some waypoints, TrajServ employs a simple map matching approach.

Figure 5.2 shows the general problem. We place a station $s_j \in S$ between two shape vertices v_i and v_{i+1} if $d_i(s_j) = (b+c)^2 - a^2$ is minimal. If s_j lies somewhere on $\overline{v_i v_{i+1}}$, d_i is always zero ($d_i = (b+c)^2 - a^2 = a^2 - a^2 = 0$). Each station s_j is sorted in between those v_i and v_{i+1} that have the smallest distance d to s_j. There are two important things to note here. First, the shape polyline we are matching the stations to is *guaranteed* to belong to this station sequence. Second, because we are respecting the order of station waypoints, each calculated position between two vertices depends not only on d, but also on every position calculated for every station before s_j. We only test s_j for positions between vertices succeeding the station vertex s_{j-1} was positioned after. After positioning s_j between two vertices, there is the possibility to project the station onto the shape (s'_1 in Figure 5.2). This usually yields smoother trajectories. Algorithm 5.1 shows the general merging procedure in pseudo-code. For each station s_j of a trajectory, we search for the best insertion place into the shape consisting of n vertices v_i. If we encounter a vertex `shape_dist_traveled` field that is bigger than the one of s_j, we insert it before this vertex. Otherwise, we compute $d_i(s_j)$ for each remaining vertex and insert s_j between those v_i, v_{i+1} where d_i is smallest. The next station s_{j+1} is guaranteed to be inserted after s_j because i is never set back to 0.

```
1:  i ← 0, v_succ ← v_0
2:  for all s_j ∈ S do
3:      d_cur ← ∞
4:      while i < n do
5:          if hasShapeDist(v_i) ∧ hasShapeDist(s_j) then
6:              if shapeDist(v_i) > shapeDist(s_j) then
7:                  break
8:              end if
9:              v_succ ← v_i ← v_{i+1}
10:             continue
11:         end if
12:         if i + 1 < n then
13:             d' = d(s_j, v_i, v_{i+1})
14:             if d' < d_cur then
15:                 d_cur = d'
16:                 v_succ = v_{i+1}
17:             end if
18:         end if
19:     end while
20:     sortIn(v_succ, s_j)
21: end for
```

Algorithm 5.1: Merging shape waypoints and station waypoints of a GTFS feed.

TrajServ already performs trivial optimizations during the loading process. If the distance of two shape vertices is below a certain threshold (usually ca. 1 m), TrajServ skips the second vertex. Similarly, if two timepoints have *exactly* the same arrival and departure times, TrajServ handles the first one as a simple shape vertex. TrajServ also translates trajectory IDs, which are allowed to be arbitrary strings in GTFS, into integers.

5.1.2. Grid Layer Construction and Indexing

After the data has been loaded into a set of trajectories, the grid layer is built. Section 3.2 gave the general idea of the data structure. In this section, we describe how a trajectory $\mathcal{T}$ is indexed, discuss the best size for a grid cell and lay out techniques that optimize the overall grid size as well as access times.

The purpose of the multi layer grid is to give fast access to trajectories that are inside the spatiotemporal bounding box B_{st} of a request. In the best case, only trajectories of vehicles that currently move through B_{st} are given to the interpolation

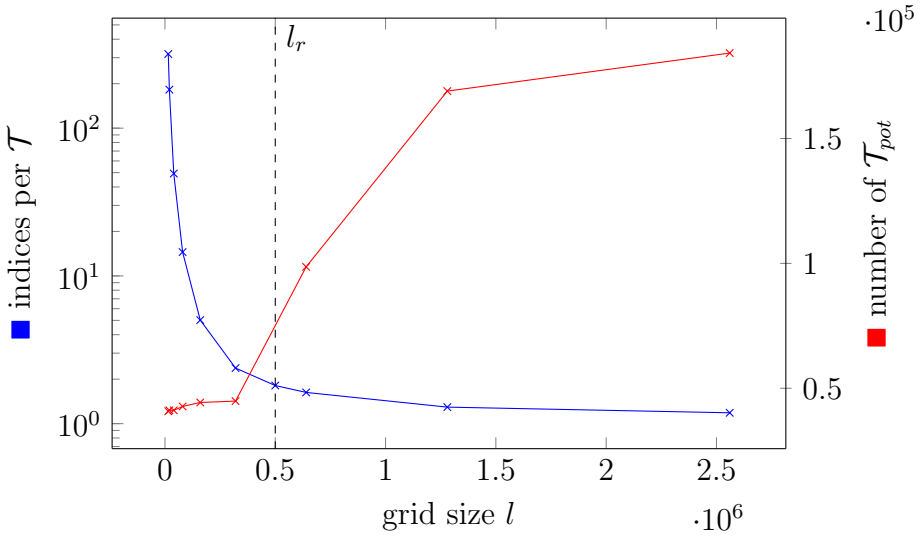

Figure 5.3: Effects of spatial grid size l. We measured the number of trajectory indices and the number of outputted potential trajectories $\mathcal{T}_{pot}$ inside a spatiotemporal bounding box with side length $l_r = 500.000$ (ca. 45 km). Data is taken from the complete Netherlands GTFS.

algorithm. In the worst case, the grid only consists of a single cell containing every trajectory of the dataset, thus reducing itself to a single, unsorted list.

To get optimal results, the side-length l of the bottom grid has to be chosen wisely. If l is too small, the grid overhead gets too large and grid lookup times exceed even the actual interpolation times. If l is too big, trajectories of vehicles that are not currently moving through the spatial part of B_{st} are given to the interpolation algorithm, resulting in unnecessary interpolations. Figure 5.3 illustrates this problem. We ran several tests against the GTFS feed of the Netherlands projected onto a 268,435,456×268,435,456 map plane. Buses, streetcars and subways were loaded into a single grid of cell size l, on which a spatial request with a square of side lengths $l_r = 500{,}000$ ($\approx$ 45km) was executed. We measured the average index count per trajectory as well as the number of trajectories that were given to the interpolation algorithm. Note that only trajectories that were active on a normal Monday were outputted. With smaller l, the number of indices per trajectory explodes. With bigger l, more and more unnecessary potential trajectories $\mathcal{T}_{pot}$ are given to the interpolation algorithm. At $l = 2.5 \times 10^6$, a single cell almost spans the whole network and the number of $\mathcal{T}_{pot}$ reaches the total number of trajectories active on a Monday. Note that the number of indices per $\mathcal{T}$ never reaches 1 because there are some trajectories that have to be indexed temporally more than one time.

Figure 5.3 also shows that in general, grid cell sizes near the average expected size of the spatial request rectangle are a good choice. However, small grid cells come at

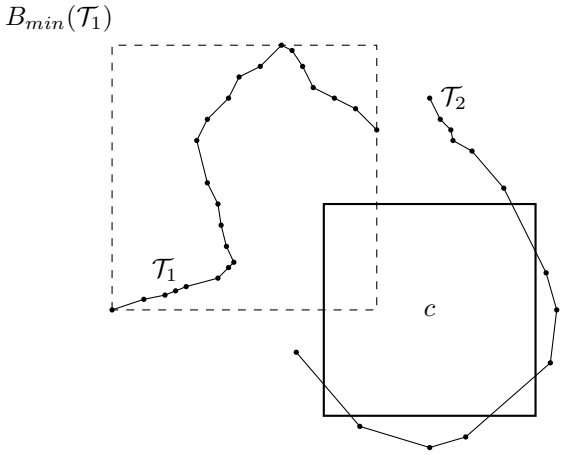

Figure 5.4: Sorting trajectories into a grid cell c.

the cost of increased memory usage.

After generating the grid layer, trajectories are indexed in two steps: first, the spatial index is computed by sorting them into grid cells. After that, they are temporally indexed by their activity function into the temporal bins of each cell.

Spatial indexing is not as trivial as it may seem at first glance. Figure 5.4 illustrates the difficulties of two naïve approaches to this problem. In the first approach, a trajectory $\mathcal{T}_1$ is spatially indexed into a grid cell c if its minimum bounding box $B_{min}(\mathcal{T}_1)$ intersects with c. As shown, this can lead to redundant indices. $\mathcal{T}_1$ never crosses c. In a different approach, $\mathcal{T}_2$ is indexed into c if one or more waypoints of $\mathcal{T}_2$ lie within c. Following this approach, $\mathcal{T}_2$ would *not* be sorted into c despite the fact that $\mathcal{T}_2$ crosses c three times.

Instead, TrajServ uses a variant of the Cohen-Sutherland clipping algorithm [15] to determine whether a trajectory really crosses a grid cell c. The algorithm surrounds the bounding box (the grid cell) with 8 rectangular regions. To efficiently calculate clipping points, flags are computed for each endpoint of a straight line that specify the region the endpoint lies in. If, for example, one endpoint lies in the upper right area and the other endpoint lies in the bottom right area, we can safely assume that the straight line does not cross the bounding box. If a nontrivial situation occurs, the algorithm clips the straight line at one endpoint based on the area the endpoint is in. In the worst case, this linear interpolation has to be done for both endpoints, which means the algorithm terminates in constant time. Cohen-Sutherland is also used in the interpolation algorithm described in the next chapter.

Section 3.1.4 introduced the concept of a trajectory's activity function α that tells for a date d if $\mathcal{T}$ is providing service on d. Activity functions are related to and created from services in a GTFS feed, but they are bit more abstract. In Section 3.2 we hinted that trajectories are indexed temporally based on α and that they are

sorted into *date bins*. We explained that besides 7 bins for each day in the week, there is a bin for trajectories that are active all day and a bin for additional services that are not provided frequently.

A trajectory is indexed into one of bins 1-8 if it is *usually* active on that day or on all workdays. This is an abstraction of the `calendar.txt` feed in GTFS. However, as in GTFS, it is still possible for a regular service to have negative exceptions (in GTFS, they are defined in `calendar_dates.txt`).

Many GTFS feeds only provide positive exceptions in `calendar_dates.txt`, which basically means that each active day of a trajectory is given explicitly. This does not allow for effective indexing and renders bins 1-8 useless. We are left with a single bin that contains each trajectory that crosses the bin's parent grid cell, *indexed for every day it is active*. In a feed that only provides `calendar_dates.txt`, we have to index each $\mathcal{T}$ per date because otherwise, for each requests, we would have to scan *all* trajectories inside a grid cell for those that are active on the time of the spatiotemporal request. This means that we either have to accept a bloated data structure or long query times, both of which are unacceptable to us. For example, consider a single trajectory that provides service from Monday till Saturday and whose service is given explicitly for each day in the GTFS feed. If the feed is valid for one year, we would have to create indices for ∼310 days T is active on, opposed to the single index in the "active on working days"-bin.

Before doing temporal indexing, TrajServ analyzes and compresses activity functions. For example, if a feed that is valid from March 3rd until March 8th 2014 provides

$$\alpha(d) = \begin{cases} 1 & \text{on 2014-03-3, 2014-03-4, 2014-03-6, 2014-03-7, 2014-03-8,} \\ 0 & \text{else} \end{cases}$$

as an activity function for $\mathcal{T}$, TrajServ counts occurrences of α on all days of the week and decides that it is more efficient to store the activity function as

$$\alpha'(d) = \begin{cases} 1 & \text{Monday till Saturdays,} \\ 0 & \text{on 2014-03-5.} \end{cases}$$

In α', $\mathcal{T}$ only has to be temporally indexed once opposed to the 6 indices α would have required.

5.1.3. Clipping and Interpolation

After trajectories have been constructed and indexed in the multilayer grid, the server enters request mode. Spatiotemporal requests as described in Section 2.2.2 are answered with unsorted lists of partial trajectories $\mathcal{T}^{par}$ that describe the clipped path a vehicle will take inside a geometrical rectangle between times t_b and t_e. If a

trajectory leaves B_{st} and reenters it multiple times, all resulting $\mathcal{T}^{par}$ are grouped by their parent trajectory $\mathcal{T}$.

In Section 3.1.4, we mentioned the problem of non-timestamped shape vertices that have to be temporally interpolated to allow for temporal clipping. We hinted that the interpolation can either be done on the fly or as a precomputation during the loading of the GTFS feed. For TrajServ, we decided to do on-the-fly interpolations. Consider again the table given at the beginning of this thesis. In the Netherlands GTFS, all trajectories contain a total sum of 34,961,174 shape vertices. In the most basic form, a waypoint timestamp consists of two POSIX-timestamps (one for arrival, one for departure time). If timestamps are stored as 32-bit integers, this would add 34,961,174 $\times$ 64bit $\approx$ 279,7 MB to the data set. If a real-time update occurs, *each* vertex of a trajectory has to hold its own arrival and departure delay fields, which adds additional 16$\times$2 bit per vertex if we want to store delays of up to ca. 1000 minutes. Additionally, the temporal interpolation would have to be recomputed for the whole trajectory on every real-time update (usually every 30 seconds) for all trajectories.

Because GTFS data already takes a lot of memory, we decided to do temporal interpolations on-the-fly while the algorithm traverses a trajectory. Thus, the clipping and interpolation algorithm has to do three basic tasks. It has to *a*) find exact (interpolated) clipping points p_b and p_e at which a trajectory enters or leaves B_{st}, *b*) output the waypoints that lie between p_b and p_e and *c*) do on-the-fly temporal interpolation to transform shape vertices into full waypoints.

The general procedure is shown in Algorithm 5.2. Input is a single trajectory $\mathcal{T}$ and a spatiotemporal bounding box B_{st}. The algorithm outputs W_{ret}, a list of all partial trajectories $\mathcal{T}^{par}$ that cross B_{st}. In natural language, it can be described as follows: we start with an empty set W_{ret} at the first waypoint of $\mathcal{T}$. We now search for the first two timepoints (here, a timepoint is a timestamped waypoint, usually station waypoints) tp_{prev} and tp_{cur} and check for entries or exits into B_{st} between them. If an entry is found, we set the entry point as p_b of the new partial trajectory, and add all waypoints until the last waypoint still in B_{st}. If this waypoint is tp_{cur}, $\mathcal{T}$ never left B_{st} between tp_{prev} and tp_{cur}. We now set $tp_{prev} = tp_{cur}$ and begin with a search between the new tp_{prev} and its new timepoint successor tp_{cur}.

For brevity, we pass on showing the `getCrossings`-algorithm. It is sufficient to know that it interpolates timestamps for shape vertices on-the-fly as described above and uses the Cohen-Sutherland algorithm to do the spatial clipping. The algorithm gets quite ugly as soon as delays are introduced. Without a depiction of the algorithm itself we state that, if `getCrossings` checks for entry or exit points into B_{st} between two waypoints p_i, p_j and $j-i = n$, the algorithm runs in $O(n)$. We have to traverse all p between p_i and p_j one time to calculate the total distance between p_i and p_j (for temporal interpolation) and another time to do the actual clipping and

1: $W_{ret} \leftarrow \emptyset$
2: $tp_{prev} \leftarrow \text{getNextTimePoint}(wp_0)$
3: $tp_{cur} \leftarrow \text{getNextTimePoint}(tp_{prev})$
4: $wp_{prev} \leftarrow tp_{prev}$
5: $\mathcal{P}^{par} \leftarrow \emptyset$
6: $p_b \leftarrow \text{null}$
7: $p_e \leftarrow \text{null}$
8: **while** $tp_{prev} \in \mathcal{P}$ **do**
9: $\quad(\mathcal{P}^{par}_{temp}, p'_b, p'_e) \leftarrow \text{getCrossings}(tp_{prev}, tp_{cur}, wp_{prev}, B)$
10: $\quad\mathcal{P}^{par} \leftarrow \mathcal{P}^{par} \cup \mathcal{P}^{par}_{temp}$
11: $\quad$**if** $p'_b \neq \text{null}$ **then**
12: $\quad\quad p_b \leftarrow p'_b$
13: $\quad$**end if**
14: $\quad$**if** $wp_{prev} \in \mathcal{P}^{par}_{temp}$ **then**
15: $\quad\quad wp_{prev} \leftarrow \text{next}(wp_{prev})$
16: $\quad$**else**
17: $\quad\quad wp_{prev} = \text{last}(\mathcal{P}^{par}_{temp})$
18: $\quad$**end if**
19: $\quad$**if** $p'_e \neq \text{null}$ **then**
20: $\quad\quad p_e \leftarrow p'_e$
21: $\quad\quad \mathcal{T}^{par} \leftarrow (\mathcal{P}^{par}, p_b, p_e, \alpha)$
22: $\quad\quad W_{ret} \leftarrow W_{ret} \cup \{\mathcal{T}^{par}\}$
23: $\quad\quad \mathcal{P}^{par} \leftarrow \emptyset, p_b \leftarrow \text{null}, p_e \leftarrow \text{null}$
24: $\quad$**end if**
25: $\quad$**if** $\text{last}(\mathcal{P}^{par}_{temp}) \geq tp_{cur} \vee tp_{cur} \notin \mathcal{P}$ **then**
26: $\quad\quad wp_{last} \leftarrow tp_{cur}$
27: $\quad\quad tp_{prev} \leftarrow tp_{cur}$
28: $\quad\quad tp_{cur} = \text{getNextTimePoint}(tp_{cur})$
29: $\quad$**end if**
30: **end while**
31: $W_{ret} \leftarrow W_{ret} \cup \{\mathcal{T}^{par}\}$

Algorithm 5.2: Basic trajectory clipping and interpolation algorithm, without delay respectation or optimizations.

interpolation. Each straight vehicle path described by two waypoints p_i and p_{i+1} is checked for temporal crossings into B_{st}, which can be done in constant time, and for spatial crossings by giving p_i, p_{i+1} and B_{st} to the Cohen-Sutherland algorithm, which calculates the clipping points in $O(1)$ [15]. Thus, `getCrossings` terminates in linear time. In Algorithm 5.2, `getCrossings` is called with a third parameter,

0		1		2		3		4		5	
δ_{arr}	δ_{dep}	δ_{arr}	δ_{dep}	δ_{arr}	δ_{dep}	δ_{arr}	δ_{dep}	δ_{arr}	δ_{dep}	δ_{arr}	δ_{dep}
0	0	**60**	60	60	**55**	55	**50**	50	50	50	50

Table 5.1: Delays for a station sequence with 6 stations. Only bold delays are actually provided in the feed.

wp_{prev} which specifies the waypoint between tp_{prev} and tp_{cur} where `getCrossings` should start. The starting point wp_{prev} is always the last waypoint `getCrossings` has outputted, which means that `getCrossings` never checks a straight line between two p_i, p_{i+1} twice. With this, we can state that the whole clipping algorithm runs in $O(n)$ with $n = |\mathcal{P}|$ being the number of waypoints of $\mathcal{T}$. It is important to note that the whole operation does not alter any data structure at all.

The waypoint coordinates of partial trajectories outputted by Algorithm 5.2 are transformed into coordinates relative to the current viewbox of the client and then outputted as JSON. Possibly the biggest effort to optimize the outputted JSON and the client performance is made during this process. For higher zoom levels, TrajServ again filters out projected waypoints (not timepoints) with predecessor distances below a certain threshold (per zoom level). This heavily optimizes the overall performance of the simulation. Without this step, TrajServ would output complete trajectories with hundreds of waypoints on zoom levels where even distances of multiple kilometers are far below the surface-width of a single projection pixel, meaning that TRAVIC would have to do thousands of interpolations without any visible movement at all.

5.1.4. Real-Time Visualization

As described in Section 2.1.3, TrajServ uses delay information to achieve real-time movements. It is able to hold an arbitrary number of GTFS-realtime feeds, fetched at regular intervals. Feeds are provided in the protocol buffer format, an efficient technique to serialize data for network communication. Protocol buffers are not self-descriptive. There is no way to tell field names or field data types from the message itself. This means that the client has to specifically compile program code that is optimized for parsing messages of a certain type (specified in a `.proto` file). Compilers are available for the most common languages [16].

Each timestamped waypoint of a trajectory can hold two delays, arrival delay δ_{arr} and departure delay δ_{dep}. In the trajectory's station sequence, a delay δ_i for station i affects all stations $> i$ and has to be explicitly neutralized by another delay (Table 5.1). After the feed has been received and parsed, TrajServ updates each affected trajectory. This is done by locking the trajectory (so that concurrent spa-

seq	0	1	2	3	4	5
δ_{arr}	0s	0s	60s	180s	240s	60s
δ_{dep}	0s	0s	120s	180s	120s	60s
t_{arr}	13:10:00	13:17:00	13:24:00	13:40:00	13:51:00	13:58:00
t_{dep}	13:11:00	13:19:00	13:25:00	13:45:00	13:52:00	13:59:00

Table 5.2: Example of erroneous delay information. The invalid delay field is marked red.

tiotemporal request runs won't produce erroneous results) and updating the arrival and delay fields of each timestamped waypoint in $\mathcal{P}$.

Both Algorithm 5.2 and the `getCrossings` algorithm described in the previous section respect delay fields by adding them to the station timestamps. Delay information is also outputted to the client. TrajServ checks each received delay update for validity, because *a*) realtime feeds sometimes output invalid delay times, especially for past stations and *b*) erroneous delay information (Table 5.2) could break the clipping algorithm completely. Table 5.2 gives an example of a corrupted delay feed. For station 4, δ_{dep} sets the departure time to 13:54:00, while δ_{arr} sets the arrival time to 13:55:00. Due to the second law of thermodynamics, this is not possible. The algorithm presented in the previous section will produce undefined results.

5.2. TRAVIC

The previous section introduced a server that can be used with any client that understands JSON. We now go on to look at the client side and present TRAVIC, a web client for TrajServ that serves both as a performance testing environment and a proof-of-concept for the combined approach described in Section 2.1.3. TRAVIC builds on Leaflet, an OpenSource JavaScript library for interactive web maps. It can handle most of the available map tile formats, though it is mostly used with Google Maps or OpenStreetMap tiles.

In this section, we describe how TRAVIC handles the data received from TrajServ, explain how the vector layer is built and discuss other possible use cases for the client. Figure 5.5 shows part of a complete TRAVIC real-time transit map of the New York/New Jersey area during the morning rush hour, including trains, subways, buses and ferries.

5.2.1. Usage of Server Data

TRAVIC fires a spatiotemporal request every sixty seconds or after the view box exceeds the bounding box of the current set of partial trajectories. After the answer was received, it iterates through every set of partial trajectories belonging to a single vehicle V, computes the current position of V (at time t_{cur}) and moves the marker accordingly. This step is repeated periodically. TrajServ uses an update interval of 50 ms at the lowest zoom level. Algorithm 5.3 shows the general approach.

```
1:  (W, B, t_b, t_e) ←doServerRequest(map.currentViewBox(), getCurTime())
2:  while simulation do
3:      B_cur ←map.currentViewBox()
4:      t ← t_0 ←getCurTime()
5:      for all W_k ∈ W do
6:          for all T_i^par ∈ W_k do
7:              if active(T_i^par, t, B_cur) then          ▷ skip T^par outside current timespan
8:                  (tp_i, tp_{i+1}) ← getCurrentTimePointPair(T_i^par, t)
9:                  p_cur ← interpolate(tp_i, tp_{i+1}, t)
10:                 if map.hasVehicleOf(T) then
11:                     move(T, p_cur)
12:                 else
13:                     draw(T, p_cur)
14:                 end if
15:             end if
16:         end for
17:     end for
18:     wait(1000/f − (getCurTime() − t_0))                ▷ respect refresh frequency f
19:     if exceeds(B_cur, t, (W, B, t_b, t_e)) then
20:         (W, B, t_b, t_e) ←doServerRequest(map.currentViewBox(), getCurrentTime())
21:     end if
22: end while
```

Algorithm 5.3: TRAVIC's main redraw algorithm (heavily simplified).

There are several important things to note here. First, since TrajServ outputs $\mathcal{T}_{par}$ as described in Section 3.1.4, TRAVIC has to do a single interpolation per partial trajectory to get the actual current position of V. This interpolation has to take place between the last waypoint before t_{cur} and the first waypoint after t_{cur}. Second, because the actual drawing of the vehicle takes up most of the computation time, TRAVIC cannot simply redraw the whole map at each interval. We describe an effective method to update the canvas in the next section. Additionally, Algo-

Figure 5.5: Real-time transit map of central New York at 8:44 EST.

rithm 5.3 leaves significant room for optimization. For example, in Line 8, it is not necessary to check the whole $\mathcal{T}^{par}$ for the current timepoints on each iteration. It is sufficient to start the lookup process at the timepoint that was last outputted as tp_i, because as mentioned before, vehicles do not travel back in time. For brevity, we left out all optimizations in Algorithm 5.3.

Because TRAVIC can both send arbitrary spatiotemporal requests to TrajServ and chose the simulation speed, it is easily possible to display time lapsed movements to visualize the vehicle movements of entire days. Due to the increasing lack of delay information over lapsed time, the simulation then converges to the static schedule.

5.2.2. The Transit Layer

The common way to display locations on web maps is to draw a marker through the map's own API. Many transit maps handle vehicles as map markers and draw them by using some map method that usually takes latitude and longitude values as coordinates. The marker is then either drawn as a single HTML element (usually an `<IMG>` wrapped inside a `<DIV>`, this approach is for example used by Google Maps or Bing) or as an actual vector object on something like an SVG layer. The latter approach is for example used by OpenLayers or Leaflet. While vector layers

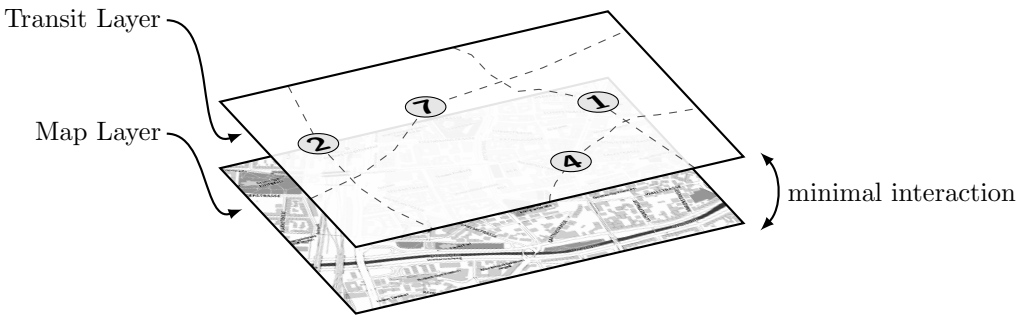

Figure 5.6: General architecture of TRAVIC and the Transit Layer.

are, in general, more efficient when it comes to displaying thousands of markers, the basic method to place markers on them is still a method accepting latitude (ϕ) and longitude (λ) coordinates as parameters. As mentioned above, this requires the map to do a projection of ϕ and λ onto the map plane, which, in this case, is the screen itself. For a few hundred markers that are positioned *once*, the projections carry no computational weight. But consider the situation depicted in Figure 5.5. During the morning rush-hour, there a very easily up to 4,000 vehicles moving on the map. Each of these is updated every 50 ms, which means that if we would do marker positioning by latitude/longitude coordinates, there would have to be $4{,}000 \times 20 = 80{,}000$ projections *per second*. In a JavaScript environment or on a mobile device, this is too much for the client to handle. This is the main reason why TrajServ projects trajectory waypoints onto the map plane and outputs only projected pixel coordinates. TRAVIC leverages this by bypassing the map service almost completely. It primarily builds on the Transit Layer, a vector layer we developed for Leaflet. It is especially designed to display vehicles of any kind moving on trajectories. Interaction between the map API (Leaflet) and the Transit Layer only consists of a few callbacks responding to map dragging or zooming. The Transit Layer is based on Raphaël, a vector library for JavaScript that makes for greater browser compatibility (there are still browsers that don't support SVG).

Figure 5.6 shows the general concept of the Transit Layer. A vector canvas is laid over the actual map layer, pans and zooms along with it and passes through DOM events like mouse clicks. Vehicles on the Transit Layer are drawn as vector objects and are positioned by pixel coordinates outputted by TrajServ that don't have to be transformed in any way.

There are still subtle difficulties in redrawing the map. For example, deleting and creating a new marker is much more expensive than re-positioning a marker that already exists on the canvas. Additionally, redrawing a marker creates a flickering effect. On the other hand, searching 4,000 markers to find the marker belonging to a single trajectory is equally expensive. TRAVIC is heavily optimized for time over

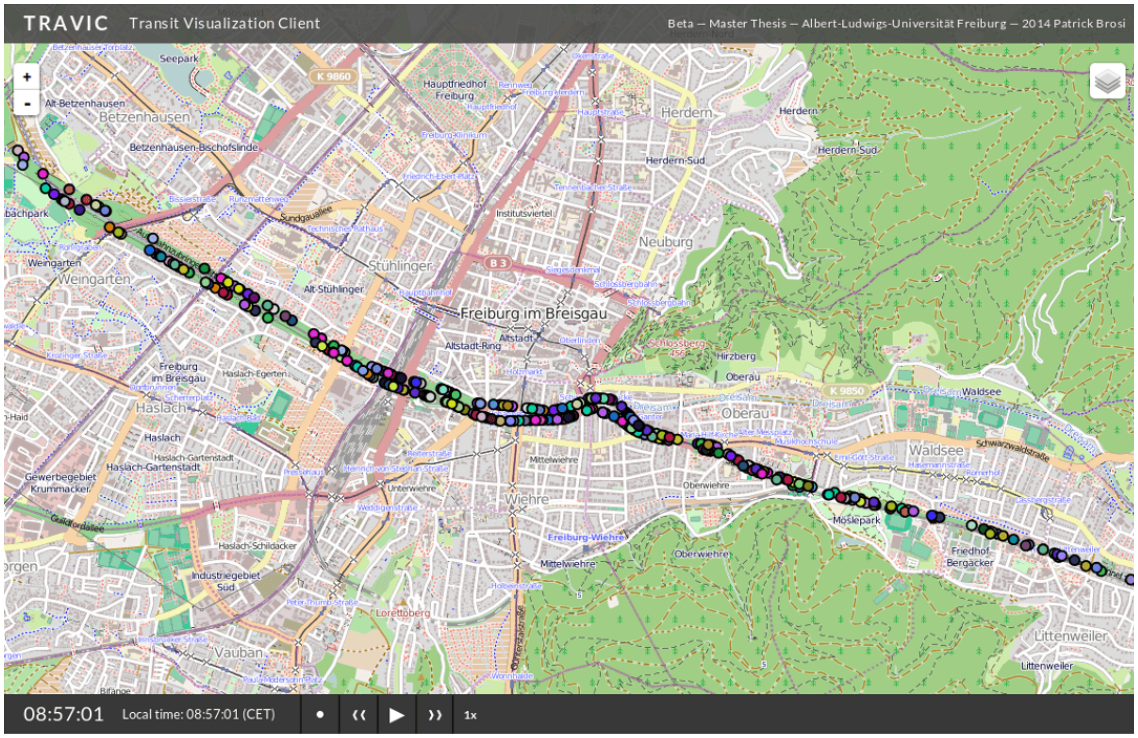

Figure 5.7: TRAVIC simulating the flow of motorized private traffic on the main east-west-corridor in Freiburg, Germany.

space and holds a simple JavaScript array containing each marker that is currently visible, indexed by their trajectory ID. Because trajectory IDs are always outputted as integers by TrajServ, most browsers implement the array as a map optimized for fast key access. Additionally, during the first update of a newly requested set of partial trajectories, the marker of each trajectory is saved as a reference field inside the trajectory object. This is just a selection of the most general speed-up techniques of TRAVIC. JavaScript optimization very quickly enters the realms of the esoteric. For brevity and spiritual salvation, we won't go into further implementation details.

5.2.3. Other Use Cases

We end this chapter with a short note on TRAVIC's applications beyond the scope of public transit. The Transit Layer can, in theory, be used to display vehicles of any kind. Outside the domain of public transportation, a vehicle could be an airplane, a satellite or even an individual object like a car, a bike or a person. A possible application of TRAVIC could be as a client for a server that outputs trajectories of randomly distributed cars traveling on certain roads to visualize the traffic volume at certain times. In the Freiburg city center, at a bridge heavily used for bike traffic,

there is a device that counts bikes passing the bridge. The numbers are publicly available [17]. Based on the numbers of a single day, we created a visualization that shows the bike traffic on the main artery through the city.

Figure 5.7 shows a visualization of the motorized private traffic on Freiburg's main east-west-corridor. The vehicle numbers follow a bimodal distribution with peaks at the morning and evening rush hours. Cars are distributed to sum up to a total number of approximately 20,000 [18] vehicles passing the central Schwabentorbrücke ($\phi = 47.990923$, $\lambda = 7.854274$) in a single direction.

```
...
{
  "id": "389045",
  "t": 3,
  "c": "#892222",
  "tc": "white",
  "hs": "Geuzenveld",
  "sn": "352",
  "ln": "Centraal Station - Geuzenveld",
  "pts": [
    [
      {
        "p": [
          206,
          265
        ],
        "at": 1385947571,
        "dt": 1385947571
      },
      [
        206,
        263
      ],
      {
        "p": [
          201,
          254
        ],
        "at": 1385947579,
        "dt": 1385947597,
        "sid": "02190"
      },
      {
        "p": [
          198,
          247
        ],
        "at": 1385947601,
        "dt": 1385947601
      }
    ]
  ]
},
...
```

Listing 5.1: Excerpt from a TrajServ JSON answer. Trajectory #3899045 to Geuzenveld has a single partial trajectory within the spatiotemporal request box, consisting of an entry point, a normal waypoint, a station waypoint and an exit point.

6. Evaluation

To evaluate the performance of TrajServ and TRAVIC, we ran tests on several GTFS feeds and measured computation times. To show that the computational cost of naïve approaches quickly grows beyond any limit reasonable for live map requests, we started with smaller networks for single cities and gradually chose bigger networks of entire countries. We then began loading TrajServ with multiple datasets from around the world. A detailed overview of the dataset parameters is given in Table 6.1. Note that the number of trajectories is counted for the entire feed, not for a specific date or time like in Table 2.1. The number of trajectories at the morning rush hour or in the late evening can be seen in the area scan results (Section 6.1). The total dataset area and the total bounding box area are equal for single GTFS feed datasets. For the combined dataset, area sums up the area sizes of the included feeds, while bounding box area is the size of the bounding box containing all feeds. Note that for country feeds, the bounding box area can be substantially larger than the country itself. This is because of international lines leaving the country.

At the time of testing, real-time feeds were available for New York, San Francisco and the Netherlands. However, delay information usually does not affect the request times at all.

6.1. Server Performance

We tested the server performance by running several spatiotemporal queries against the datasets described above. Each query requested the partial trajectories for the

GTFS feed	#trajectories	#stops	#arr/dep	#vertices	area	bbox area
Vitoria-Gasteiz	6,041	338	122,184	4,198	66.84	66.84
Budapest	147,556	5,357	2,660,027	237,386	1,952	1,952
New York Area	300,417	34,948	11,665,443	3,482,713	98,965	98,965
Switzerland	138,462	21,689	2,092,196	-	$1.3 \cdot 10^6$	$1.3 \cdot 10^6$
Netherlands	548,007	73,293	12,221,953	3,843,780	$2.5 \cdot 10^7$	$2.5 \cdot 10^7$
Combined feed	2,507,566	298,535	76,287,281	12,147,015	$\sim 5 \cdot 10^7$	$\sim 10^8$

Table 6.1: Datasets used for testing. Areas are given in km^2.

next 60 minutes and was run twice: once in the morning rush hour with t_b = 8:00:00 and once in the late evening with t_b = 23:00:00. In a first step, we did a total area scan that requested the partial trajectories of the entire dataset ($z = 20$). We then proceeded with a spatiotemporal bounding box request that resembles the requests fired by real clients. The request box with area $A \approx 10$ km^2 was handpicked from the dataset's center. The request zoom level was $z = 20$. A third request covered an area of about 60,000 km^2 at zoom level $z = 9$, which is roughly the size of Switzerland's bounding box.

All of these queries were answered using three different approaches. First, we ran the test using a naïve approach where trajectories were stored as an unsorted list. We then did the same request on a layered grid with base cell side length l = 500,000 (approximately 40 km in central Europe.) A third test run used a layered grid with base cell side length l = 250,000. Note that in the naïve approach, there are still some very simple optimizations present. Consider, for example, a query that requests all partial trajectories between 8:00 and 9:00. If the naïve approach finds a trajectory that leaves the first station at 9:20 and arrives at the last station at 10:50, the trajectory is skipped and Algorithm 5.2 is not called.

We measured the total query time, the number of outputted partial trajectories and the number of *affected* trajectories. We say a trajectory $\mathcal{T}$ is affected during a query if a non-trivial computation has to be executed on $\mathcal{T}$. Non-trivial computations include, for example, Algorithm 5.2 and a call to the trajectory's activity function. Calls to getter functions are considered trivial. A good measurement for the scalability of an approach is the ratio of affected trajectories and outputted partial trajectories. If, for example, TrajServ outputs 500 partial trajectories and only had to look at 700 trajectories at all, we consider this a good result.

All tests were executed on a machine with two Intel Xeon E5640 CPUs (8 cores in total) and 66 GB of RAM. TrajServ was compiled with `gcc` 4.4.6 and optimization level `-O3`. For the layered grid queries, the vehicle types displayed at different zoom levels can be seen in Table 6.2. The results in Tables 6.1 to 6.9 are averaged values from 100 sample runs.

type	streetcar	subway	rail	bus	ferry	cable car	gondola	funicular
z	13-20	11-20	5-20	14-20	5-20	11-20	11-20	11-20

Table 6.2: Vehicle types on different zoom levels in the testing scenario.

For small networks like Vitoria-Gasteiz, the naïve approach yields reasonable good results. However, even for this small dataset, request times are 5 to 20 times higher than for the grid layer approach. Table 6.3 shows that, despite the fact that Vitoria-Gasteiz does not have any vehicles that are displayed at zoom level 9, the naïve approach still has to check 6,000 trajectories for the $z = 9$ box request.

For bigger datasets like the Swiss rail network or Budapest, box requests at ground level are still 20 times faster with the grid layer than with the naïve approach (Tables 6.5 and Table 6.6). We found that for total area scans, a bigger grid size usually yields better results. This is due to the computational overhead costs of the grid. For box requests, the $l = 250{,}000$ grid yields only insignificant speed up, which again can be explained with overhead costs.

In Table 6.8, the $z = 9$ box request for the New York City area at 23am yields 161 partial trajectories. To output this (small) number of $\mathcal{T}^{par}$, the naïve approach has to look at 300,000 trajectories, while the $l = 250{,}000$ grid only has to look at 227.

Similar results can be seen in Table 6.7. At the time of writing this thesis and to our best knowledge, the Netherlands GTFS is (by far) the biggest transit feed available. It can be considered as "complete", meaning that *every* public transit vehicle in the country is included with its full polyline. Still, despite the fact that the total number of trajectory vertices is 16 times as high as in the Budapest feed (the number of arrival/departure events is nearly 5 times as high), the time to output ca. 1,000 partial trajectories at ground level in the city center of Amsterdam is only slightly bigger than the time it took to output 1,000 $\mathcal{T}^{par}$ in the city center of Budapest. The higher time for the $l = 250{,}000$ box request in Amsterdam can be explained by the higher network density in the city (2,000 affected trajectories vs. 1,500 affected trajectories in Budapest) and mainly because of the higher vertex density in the Netherlands feed. Note that the $z = 9$ box requests in Table 6.7, which nearly covers the whole country of the Netherlands, only takes 23 ms during the morning rush hour. However, keep in mind that at this zoom level, only trains are outputted by TrajServ (Table 6.2).

In the introduction, we mentioned that TrajServ is easily able to handle the GTFS feed of entire countries. We consider this statement proved by Tables 6.6 and 6.7. To show that TrajServ is able to handle the public transit network of the whole world, we ran the query of Table 6.7 against a dataset consisting of 22 feeds from around the world. These feeds include three entire countries (Switzerland, Sweden and the Netherlands). The other feeds are: public transit in the cities of Albuquerque, Boston, Los Angeles, Miami, San Francisco, Portland, Chicago (all USA), Québec, Montreal (both Canada), Manchester (UK), Budapest (Hungary), Rennes (France), Turin (Italy), Vitoria-Gasteiz (Spain), Auckland, Wellington (both New Zealand), Adelaide (Australia) and the public transit in the areas of Freiburg (Germany) and New York (USA). The parameters of this combined feed are listed in Table 6.1.

Table 6.9 shows that even when run against the combined dataset, the grid layer approach is still able the handle requests very fast. For the box requests, computation times are nearly the same (± 1 ms).

6.2. Client Performance

Measuring JavaScript Performance is a difficult task. Code examples that run efficiently on one browser type can completely lock up another one. To evaluate the performance of TRAVIC, we chose to run tests on the current versions of five different web browsers: Firefox 27.0, Internet Explorer 11.0.2, Chromium 32.0.1700, Safari 5.1.7 and Opera 12.16. We loaded a TrajServ installation with the combined feed described in the previous section, centered TRAVIC at the intersection of transept and nave of the "Oude Kerk" in Amsterdam ($\phi = 52.374359$, $\lambda = 4.898161$) at the highest zoom level possible and gradually zoomed out. For each zoom level, we measured the number of displayed vehicles $\#v$ and the time it took to do a single screen refresh t_r. We did the same tests starting at the dome of New York City Hall ($\phi = 40.712737$, $\lambda = -74.005973$). Since TRAVIC uses lower refresh rates ($1/f$) at lower zoom levels, we multiplied this time with the number of refreshes per second at each level to get the amount of time t_{tot}/s TRAVIC was busy with refreshing the screen during a single second. If $t_{tot}/\text{s} > 1000$ ms, TRAVIC could not compute the intended number of refreshes in one second.

Tests for Firefox, Google Chrome and Opera were done on a machine with an Intel Core i5-3320M Processor, 8 GB RAM and NVIDIA Quadro NVS 5400M graphics, running Ubuntu 13.10. Tests for Internet Explorer and Safari were done on the same machine, running Microsoft Windows 8 (SP1). All tests were run in full-screen mode at 1920×1080. Values are averaged from 100 sample runs. Results can be seen in Tables 6.10 and 6.12.

6.3. Asynchronous Delay Information

We end this chapter with a specific remark on the accuracy of TRAVIC. A problem that is inherent to the approach of combining static schedule data and real-time delay information into partial trajectories that are sent to the client is asynchronous delay information. Consider the scenario depicted in Figure 6.1. A vehicle travels on a certain route between two waypoints. At time t_1, the client sends a spatiotemporal request to the server, which currently holds a delay $\delta = 5$ for the trajectory. A delay of 5 is communicated to the client. Now, the server does an update and fetches the newest version of the real-time feed. Somehow, the vehicle has regained the lost time and δ is now 0 (at $t = t_2$). The client, however, still operates within the bounds of the spatiotemporal request answer received at t_1. No new request is made. Thus, for the client, δ is still 5. Now, at $t = t_3$, the client finally fires a new spatiotemporal requests and learns that the actual position of the vehicle is far behind the position it currently displays. There is, of course, also the possibility of the vehicle being far ahead of the current client position.

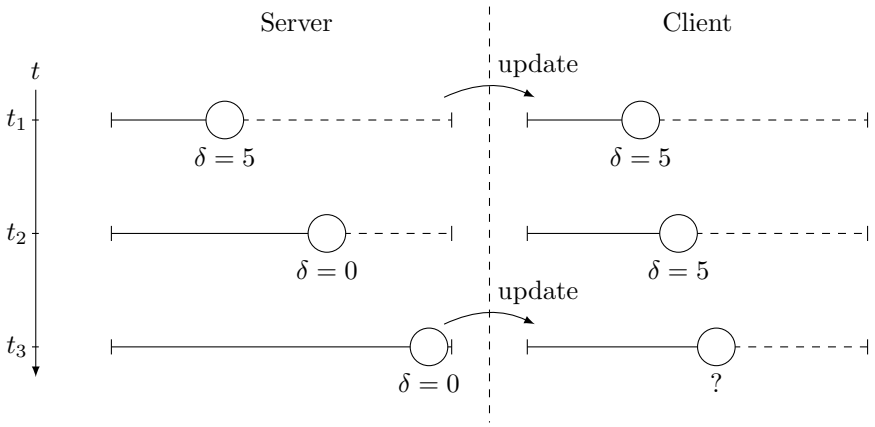

Figure 6.1: Problems with asynchronous delay information.

There are two possible outcomes of this situation. In the first one, the client tries to be as accurate as possible and immediately corrects the position, resulting in a vehicle that jumps ahead (possibly multiple kilometers for long distance trains). In the second, the client prioritizes smooth vehicle movements. If the first waypoint p_1 of the new partial trajectory is *ahead* of the current client position p_{cur}, the travel speed is calculated in regard to p_{cur} and an additional interpolation takes place between p_{cur} and p_1. The vehicle will move faster to the next station, but it will not jump. If p_1 is *behind* p_{cur}, p_1 is completely ignored and travel speed is again calculated in regard to p_{cur}. The vehicle will move slower to the next station.

Currently, TrajServ displays vehicle jumps, but this remains an open issue.

	Naïve		Grid ($l = 500$k)		Grid ($l = 250$k)	
	8am	11pm	8am	11pm	8am	11pm
Total area scan						
time in ms	18	17	3	< 1	3	< 1
#partial trajectories	175	71	175	71	175	71
#affected trajectories	6 k	6 k	277	126	277	126
request area in km^2	66.8	66.8	66.8	66.8	66.8	66.8
Box request						
time in ms	17	17	3	< 1	3	< 1
#partial trajectories	188	76	188	76	188	76
#affected trajectories	6 k	6 k	277	125	277	125
request area in km^2	9.9	9.9	9.9	9.9	9.9	9.9
Box request with $z = 9$						
time in ms	12	12	< 1	< 1	< 1	< 1
#partial trajectories	0	0	0	0	0	0
#affected trajectories	6 k	6 k	0	0	0	0
request area in km^2	51.5 k	51.5 k	51.5 k	51.5 k	51.5 k	51.5 k

Table 6.3: Testing results for Vitoria-Gasteiz.

	Naïve		Grid ($l = 500$k)		Grid ($l = 250$k)	
	8am	11pm	8am	11pm	8am	11pm
Total area scan						
time in ms	15	13	4	< 1	4	< 1
#partial trajectories	4,3 k	768	4,3 k	768	4,3 k	768
#affected trajectories	45 k	45 k	4,3 k	768	4,3 k	768
request area in km^2	38	38	38	38	38	38
Box request						
time in ms	15	13	4	< 1	4	< 1
#partial trajectories	4,3 k	768	4,3 k	768	4,3 k	768
#affected trajectories	45 k	45 k	4,3 k	768	4,3 k	768
request area in km^2	10.1	10.1	10.1	10.1	10.1	10.1
Box request with $z = 9$						
time in ms	7	7	< 1	< 1	< 1	< 1
#partial trajectories	0	0	0	0	0	0
#affected trajectories	45 k	45 k	0	0	0	0
request area in km^2	60 k	60 k	60 k	60 k	60 k	60 k

Table 6.4: Testing results for simulated traffic in Freiburg.

	Naïve		Grid ($l = 500$k)		Grid ($l = 250$k)	
	8am	11pm	8am	11pm	8am	11pm
Total area scan						
time in ms	500	480	100	50	102	51
#partial trajectories	3.3 k	1.6 k	3.3 k	1.6 k	3.3 k	1.6 k
#affected trajectories	147.6 k	147.6 k	3.8 k	1.8 k	3.8 k	1.8 k
request area in km^2	1.9 k	1.9 k	1.9 k	1.9 k	1.9 k	1.9 k
Box request						
time in ms	457	460	27	12	27	12
#partial trajectories	1 k	434	1 k	434	1 k	434
#affected trajectories	147.6 k	147.6 k	1.5 k	628	1.5 k	628
request area in km^2	9.9	9.9	9.9	9.9	9.9	9.9
Box request with $z = 9$						
time in ms	328	326	1.8	< 1	1.8	< 1
#partial trajectories	73	39	73	39	73	39
#affected trajectories	147.6 k	147.6 k	73	39	73	39
request area in km^2	60.2 k	60.2 k	60.2 k	60.2 k	60.2 k	60.2 k

Table 6.5: Testing results for Budapest.

	Naïve		Grid ($l = 500$k)		Grid ($l = 250$k)	
	8am	11pm	8am	11pm	8am	11pm
Total area scan						
time in ms	420	420	50	25	56	30
#partial trajectories	6.4 k	2.8 k	6.4 k	2.8 k	6.4 k	2.8 k
#affected trajectories	138 k	138 k	7 k	3.1 k	7 k	3.1 k
request area in km^2	1.3 M	1.3 M	1.3 M	1.3 M	1.3 M	1.3 M
Box request						
time in ms	410	410	30	18	28	17
#partial trajectories	411	242	411	242	411	242
#affected trajectories	138 k	138 k	539	330	539	330
request area in km^2	10.2	10.2	10.2	10.2	10.2	10.2
Box request with $z = 9$						
time in ms	320	320	4.2	2.2	4.5	2.5
#partial trajectories	875	434	875	434	875	434
#affected trajectories	138k	138 k	959	505	959	505
request area in km^2	60 k	60 k	60 k	60 k	60 k	60 k

Table 6.6: Testing results for Switzerland.

	Naïve		Grid ($l = 500$k)		Grid ($l = 250$k)	
	8am	11pm	8am	11pm	8am	11pm
Total area scan						
time in ms	2.1 k	1.9 k	706	310	998	434
#partial trajectories	11.5 k	5.1 k	11.5 k	5.1 k	11.5 k	5.1 k
#affected trajectories	548 k	548 k	18.1 k	8.2 k	18.1 k	8.2 k
request area in km^2	25 M	25 M	25 M	25 M	25 M	25 M
Box request						
time in ms	1.82 k	1.82 k	51	33	50	31
#partial trajectories	911	556	911	556	911	556
#affected trajectories	548 k	548 k	2 k	1.2 k	2 k	1.2 k
request area in km^2	10.7	10.7	10.7	10.7	10.7	10.7
Box request with $z = 9$						
time in ms	1.28 k	1.28 k	23	14	23	14
#partial trajectories	707	451	707	451	707	419
#affected trajectories	548 k	548 k	986	533	986	533
request area in km^2	60 k	60 k	60 k	60 k	60 k	60 k

Table 6.7: Testing results for the Netherlands.

	Naïve		Grid ($l = 500$k)		Grid ($l = 250$k)	
	8am	11pm	8am	11pm	8am	11pm
Total area scan						
time in ms	1.3 k	1.1 k	478	155	495	185
#partial trajectories	11.5 k	3.6 k	11.5 k	3.6 k	11.5	3.6 k
#affected trajectories	300 k	300 k	17.2 k	3.8 k	17.2 k	3.8 k
request area in km^2	98 k	98 k	98 k	98 k	98 k	98 k
Box request						
time in ms	1.1 k	1 k	87	26	83	25
#partial trajectories	1.1 k	353	1.1 k	353	1.1 k	353
#affected trajectories	300 k	300 k	3.4 k	783	3.4	783
request area in km^2	10.7	10.7	10.7	10.7	10.7	10.7
Box request with $z = 9$						
time in ms	676	670	13	5	13	5
#partial trajectories	485	161	485	161	485	161
#affected trajectories	300 k	300 k	559	227	559	227
request area in km^2	60.8 k	60.8 k	60.8 k	60.8 k	60.8 k	60.8 k

Table 6.8: Testing results for New York and New Jersey.

	Naïve		Grid ($l = 500$k)		Grid ($l = 250$k)	
	8am	11pm	8am	11pm	8am	11pm
Total area scan						
time in ms	8 k	9 k	1.3 k	1.7 k	1.7 k	1.9 k
#partial trajectories	40.5 k	40.3 k	40.5 k	40.3 k	40.5 k	40.3 k
#affected trajectories	2,5 M	2,5 M	67.7 k	67.1 k	67.7 k	67.1 k
request area in km^2	∼100 M	∼100 M	∼100 M	∼100 M	∼100 M	∼100 M
Box request						
time in ms	7.5 k	8.2 k	51	33	50	32
#partial trajectories	911	556	911	556	911	556
#affected trajectories	2,5 M	2,5 M	2 k	1.2 k	2 k	1.2 k
request area in km^2	10.7	10.7	10.7	10.7	10.7	10.7
Box request with $z = 9$						
time in ms	5.9	5.9	23	15	23	15
#partial trajectories	707	451	707	451	707	451
#affected trajectories	2,5 M	2,5 M	986	533	986	533
request area in km^2	60 k	60 k	60 k	60 k	60 k	60 k

Table 6.9: Testing results for the combined feed. Box request areas are the same as in Table 6.7. Request times are CET.

			Chrome		Firefox		Opera		Safari		IE	
z	$1/f$	$\#v$	t_r	t_{tot}/s	t_r	t_{tot}/s	t_r	t_{tot}/s	t_r	t_{tot}/s	t_r	t_{tot}/s
19	60	0	< 0.1	0.5	< 0.1	1.2	< 0.1	1.3	< 0.1	0.8	< 0.1	0.7
18	85	3	4.7	55.8	7.2	84.8	6	70.1	6.6	77.5	7.8	91.9
17	110	33	16	139.4	21.5	195.8	17.7	160.6	10.5	95.7	33.2	301.4
16	120	74	13.6	113.2	22.6	188.7	15.9	132.6	11.7	97.7	28.9	240.5
15	140	145	24	171.3	35.0	250.1	27.0	193.1	19.3	137.5	50.6	361.6
14	170	317	35	205.8	56.4	331.9	35.1	206.5	29.6	174.3	77.4	455.2
13	250	200	31.5	126.1	48.3	193	37.8	151.2	25.7	102.7	94.7	378.9
12	400	95	38	94.9	57.7	144.3	50.5	126.3	34.1	85.3	216.8	542.1
11	500	130	15.4	30.7	17.4	34.8	15.2	30.5	8.7	17.4	18.6	37.1
10	1 k	221	20	20	27.2	27.2	19.8	19.8	14.9	14.9	2.2	24.2
9	1 k	363	27.6	27.6	38.3	38.8	26.2	26.2	19.4	19.4	34.6	34.6
8	1 k	423	33.9	33.9	45.5	45.5	31.6	31.6	23.3	23.3	38.9	38.9
7	1 k	455	32.6	32.6	47.2	47.2	32	32	24.9	24.9	40.8	40.8
6	1 k	1 k	69.2	69.2	109.4	109.4	33.1	33.1	47.2	47.2	88	88
5	1 k	1.2 k	85.7	85.7	124.4	124.4	40.3	40.3	64.3	64.3	104.2	104.2

Table 6.10: TRAVIC performance on different browser types and zoom levels. Map was centered at Amsterdam.

			Chrome		Firefox		Opera		Safari		IE	
z	$1/f$	$\#v$	t_r	t_{tot}/s	t_r	t_{tot}/s	t_r	t_{tot}/s	t_r	t_{tot}/s	t_r	t_{tot}/s
19	60	30	2.6	42.6	3.1	52.3	2.5	41.7	1,8	29.8	4.2	70.2
18	85	68	5.2	60.6	5.7	67.5	4.3	50	4.3	50.4	9.3	168.9
17	110	117	6.5	59.4	7.9	71.8	5.9	54	6.4	58.1	14.3	129.6
16	120	195	9.2	76.5	11.3	94.1	8.2	67.9	10.3	85.7	20.5	171.1
15	140	285	11.2	80.2	14.1	100.9	10.7	77.6	12.5	89.2	26.1	185.9
14	170	331	12.7	74.8	15.5	91.4	11.8	69.9	13.9	81.7	26.5	175.9

Table 6.11: TRAVIC performance on different browser types and zoom levels. Map was centered at Freiburg and individual traffic was simulated on the main corridor.

			Chrome		Firefox		Opera		Safari		IE	
z	$1/f$	$\#v$	t_r	t_{tot}/s	t_r	t_{tot}/s	t_r	t_{tot}/s	t_r	t_{tot}/s	t_r	t_{tot}/s
19	60	4	2.6	43.5	3.5	57.8	3.3	54.5	3.6	59.8	9.9	165.3
18	85	14	6.4	75.2	8.3	98.1	7.2	85.2	9.9	116.8	28.8	332.7
17	110	47	8.7	79.1	10.4	94.1	9.3	84.4	13.3	121.3	34.1	310.1
16	120	111	17.3	144.3	24.3	202.5	19.9	165.5	25.8	215.3	135.9	1.1 k
15	140	233	30.4	217.4	45.6	325.5	38	271.6	46.8	334	458.8	3.3 k
14	170	745	32.3	189.7	55.5	326.2	34.9	205	44.5	261.9	172.1	1 k
13	250	431	23.1	92.4	34.9	139.5	25.9	103.8	35.1	140.4	142	568.1
12	400	599	33.3	83.3	53.3	133.3	34.9	87.4	48.2	120.6	200.4	500.9
11	500	674	21.5	43	34.1	68.2	24	48	34.3	68.5	55	110
10-5	1 k	210	10.8	10.8	13	13	9.9	9.89	16.6	16.6	25.2	25.2

Table 6.12: TRAVIC performance on different browser types and zoom levels. Map was centered at New York.

7. Future Work

At the time of writing this thesis, TrajServ and TRAVIC are stable and have been heavily tested for weeks. Chapter 6 showed that performance of both the client and the server is very good. Still, we think there is potential for improvements. During the work on this thesis, we also came up with ideas for additional features and encountered some unexpected problems. The first section of this chapter serves as a collection and an outline of future work related to real-time public transit visualization. In the second section, we address the more complex problem of extracting vehicle routes from geospatial datasets using map matching and route planning strategies.

7.1. Improvements and Additional Features

Vehicle Trajectory Optimization Shape vertices use a lot of memory space and slow down the clipping/interpolation algorithm. We mentioned in Section 5.1.1 that TrajServ filters out waypoints whose distance is below a certain threshold. After applying this optimization, there are still many shape vertices that are completely redundant. Figure 7.1 gives examples of such waypoints. We figure that an algorithm to reduce the number of shape vertices in a trajectory could lead to significant speedup and to less memory usage. It could be feasible to apply the Ramer-Douglas-Peucker [19] algorithm to the dataset before storing it into the grid layer. Since RDP restricts the vertices of the simplified piecewise linear curve to a subset of the original ones, the results are not optimal. There are other algorithms that approximate a given piecewise linear curve with the minimum number of vertices required to stay below a certain error bound [20].

Route Planner Integration One of the most interesting features we would like to implement in the future is the integration of TrajServ and TRAVIC into a route planning environment. Our goal is to create a full-featured public transit applica-

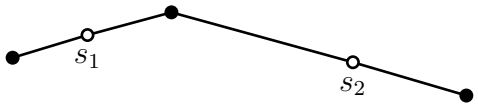

Figure 7.1: Redundant vertices s_1, s_2 on a piecewise linear curve.

tion that is not only able to provide route planning services, but can also display the live positions of the vehicles a user is supposed to take for a given route or currently sits in. TrajServ can already handle requests for current positions of single trajectories, so the main task would be to create a working client. One idea could be to combine the Multimodal Routeplanner Client [21] with TRAVIC.

Asynchronous Delay Information As mentioned in Section 6.3, the problem of delay asynchronicity still remains an open issue.

Mobile Client We plan to translate the JavaScript code of TRAVIC into Java code to provide a mobile application for Android devices.

Map Tile Server It would be useful beyond the scope of live transit maps to provide schematic network map tiles that are automatically rendered from the GTFS. To our knowledge, no such tile server exists at the moment. To render a static map of the dataset, TrajServ would have to build a graph, analyze the number, types and labels of each vehicle connecting two stations and render the link between those stations accordingly.

Memory Usage TrajServ still uses a lot of memory. Much of the memory consumption is due to the fact that each trajectory holds references to each of its shape vertices. This could be optimized by storing "chunk references" of waypoints. So, instead of storing references to shape vertices `4545, 4546, 4547, 4548, 4549, 4550, 4551, 4552` between two stations, it would be better to store `<4545 - 4552>`.

Support of Frequencies The GTFS standard includes vehicles that are not served according to an exact schedule, but with a certain frequency at different times (like the Paris Metro). Neither TrajServ nor TRAVIC are currently able to consider trips that are defined in this way.

Support of Blocks To integrate multiple trips into a single vehicle (for example to model vehicles that branch at a certain station into two separate trips) GTFS provides the concept of blocks. At the moment, TrajServ completely ignores the `block_id`. In the future, this field should be used by both the server and the client.

GTFS Output During the development of the server, it became clear to us that TrajServ is by now a powerful tool to validate and minimize GTFS feeds. The GTFS parser can handle corrupted feeds, is able to optimize the number of shape vertices, dramatically reduces the number of service dates by transforming explicit service dates into weekly services with exceptions and can even add missing `shape_distance_travelled` fields. To harvest this functionality for other areas of public transit, TrajServ should be able to transform its internal representation of trajectories back into the GTFS format.

7.2. Extraction of Vehicle Routes

One of the biggest drawbacks of the increasing availability of public transit feeds is the fact that many of them lack exact trajectory shapes. Without proper polylines of a trajectory, vehicle routes are presented as straight station-to-station connections to the end user. While this is a minor problem for bus and streetcar networks with a high station density, long-distance trains that travel hundreds of kilometers without a single stop appear to by flying over the countryside in the live map. It seems to us that part of the problem is that many public transit companies simply do not have exact spatial data of their vehicle routes. If there is any information at all (like exact geospatial data of a rail network), it is not connected with single vehicle trips. Because of this, it is necessary to either associate trajectory shapes by hand or to extract the routes from geospatial datasets.

In this section, we roughly sketch several ideas of how to do such an extraction. This has applications beyond the scope of public transit movement visualization. The data can be used by transit companies internally or by route planners to output the exact path a vehicle will take. We will discuss two entirely different approaches: classic map matching and route planning.

7.2.1. Map Matching

Map matching is the process of associating geographical positions to map data. It is for example used in route planning systems to tell the exact road a car is currently on or to "snap" paths of collected GPS data (with possible measurement errors) to actual roads. We found very little literature on applying map matching algorithms to extract public transit trajectories. Brakatsoulas et al. hint that most map matching algorithms are optimized to find the current location of a single GPS coordinate [22]. They give an overview of several approaches to map *lists* of geographical positions to map data. Although the primary focus of Brakatsoulas et al. is on mapping collected GPS data from cars, we argue that their approaches can also be applied to vehicle route extraction. A trajectory's (ordered) list of station waypoints can be understood as a set of (very sparse) sample positions along the way of the vehicle.

We hint that we already gave a subtle description of matching station waypoints to their exact route shape. In Section 5.1.1, we presented an algorithm that sorted station waypoints into shape vertices during the loading process of the GTFS feed. This algorithm works as follows: for each station waypoint s_j, we look at all vertices v_i *after* the vertex v_{in} the station waypoint s_{j-1} was inserted at. For each pair of vertices proceeding v_{in} (for each edge e), we calculate a distance δ between e and s_j. After all distances have been computed, we chose the edge with minimal δ as the new edge to hold s_j. This algorithm can be optimized into a greedy algorithm

that stops lookaheads if d gets bigger with each step (such reducing the probability of s_j being inserted into any proceeding edge). TrajServ uses the distance function

$$\delta_i(s_j) = (b + c)^2 - a^2, \tag{7.1}$$

where b is the distance between v_i and s_j, c is the distance between v_{i+1} and s_j and a is the length of edge (v_i, v_{i+1}) (Figure 5.2).

In Section 5.1.1, the shape path we associated the set of stations to was always guaranteed to belong to the stations. This is no longer the case in map matching. More importantly, paths can now have branches. However, we argue that the approach still applies. Consider the algorithm tries to match the station waypoints of a train to a railroad track. If the track branches, we simply run the algorithm on all branches and chose the one that has the minimum (averaged) δ. Brakatsoulas et al. describe a similar algorithm (with another distance formula) that is limited to exactly 4 lookahead steps per branch [22]. After 4 edges of a branch have been checked, the next branch is looked at. Distances can either be compared by the maximum distance per branch or by the average distance per branch. Because of the large distance between sample points, however, a limitation of 4 edges does not seem practical to us for transit route extraction. Another possibility to avoid the explosion of computation time in huge networks is the approach described above where lookahead stops if the distance only gets bigger with each step. This can lead to false results for complicated vehicle paths. In some cases, the geospatial dataset holds information that can be used to sort out entire branches very easily. Usable information includes lists of vehicle types for each branch or even lists of route description strings that can be compared to the route descriptions stored in the public transit feed.

Another strategy is to do global matching. Global matching tries to find a path inside the geospatial network data that is as close as possible to the path described by the list of sample points [22]. Comparison between two paths can be done using the Fréchet Distance δ_F.

We mentioned in Section 3.1.4 that a vehicle trajectory can be understood as a parametrization of a piecewise linear curve in $\mathbb{R}^2$ (Definition 4). There, the pa-

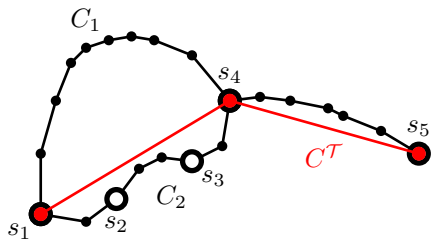

Figure 7.2: Problems with map matching algorithms for vehicle trajectories.

rameter was time t. We can also chose the percentage p of traveled distance as a parameter, thus yielding a parametrization $[0,1] \mapsto \mathbb{R}^2$. The same applies to any path in the geospatial network data.

The Fréchet Distance is commonly described as the minimum length a leash has to have to connect a dog and its owner, both traveling on separate paths A and B.

More formally:

$$\delta_F(A,B) = \inf_{\alpha,\beta} \max_{p\in[0,1]} \left\{ d\big(A(\alpha(p)),\, B(\beta(p))\big) \right\}, \tag{7.2}$$

where α and β are non-decreasing mappings $[0,1] \mapsto [0,1]$ and d is the distance function in $\mathbb{R}^2$. Because α and β are non-decreasing, it is not possible for dog or owner to travel backwards.

While the Fréchet Distance is one of the best similarity measures for curves, there are some problems when it comes to route extraction for public transit vehicles. The main problem is the low sample rate induced by the fact that only station position are available for a trajectory curve. Figure 7.2 gives an example of a simple railroad network consisting of two paths C_1 and C_2. $C_\mathcal{T}$ is the piecewise linear curve induced by a trajectory $\mathcal{T}$. The Fréchet Distance method (and the local method described above as well) yields C_2 as the curve closest to $C_\mathcal{T}$. But what if $C_\mathcal{T}$ is actually a long distance rapid train that takes the high speed track C_1 between stations s_1 and s_4 to avoid driving through all stations between s_1 and s_4?

We suggest to use a slightly altered version of the Fréchet Distance that takes into account the number of skipped stations. If A is a curve induced by a trajectory $\mathcal{T}$, B is a curve in a geospatial public transit dataset, S is the set of stations on A and S' is the set of stations on B, we propose a distance function like

$$\delta'_F(A,B) = \delta_F(A,B) + c_1\big|S' \setminus S\big| + c_2\big|S \setminus S'\big|. \tag{7.3}$$

Constant c_1 is the penalty for skipped stations, c_2 is the penalty for stations that are on A, but do not appear on B.

7.2.2. Route Planning

Map matching can be a reasonable strategy for low-density, rail-bound transit networks. Bus routes, however, use the road network. Even between two single bus stops, there could be thousands of possible routes for the bus to take (Figure 7.3). Classic map matching does not seem feasible here.

It is adequate to assume that between two stops, a bus always takes the shortest route. A robust solution to get the complete route a bus takes through the road network could be to do shortest-path queries between each consecutive stops. The

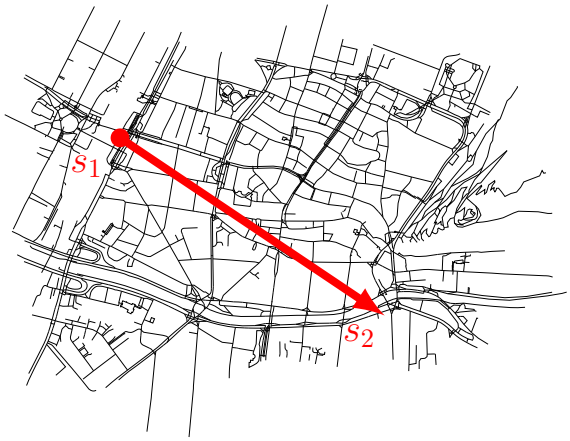

Figure 7.3: Direct connection.

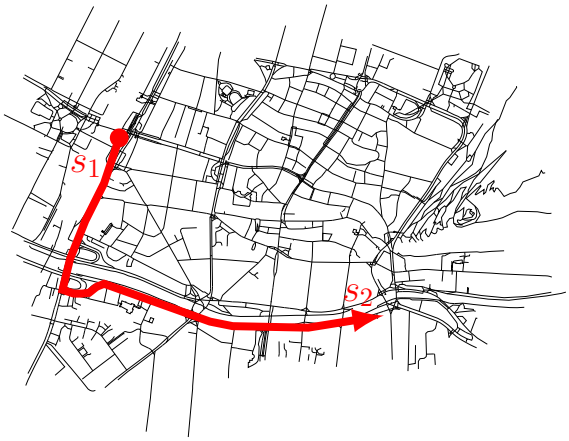

Figure 7.4: Shortest path.

shortest-path algorithm would have to respect roads that are closed for buses as well as special bus-only roads. This approach could also be applied to rail-bound networks.

We presume that routes computed by iteratively computing shortest paths between stops are very close to the actual routes the vehicles take. In Figure 7.4, the shortest path between two regional bus stops in Freiburg (computed with the Google Directions API) is the same as the actual route. Even if they slightly differ, the computed routes can still be used as a very good starting point to do manual corrections.

8. Conclusion

This thesis presented a scalable approach to do live visualizations of real-time public transit data. We described a strategy that handles vehicle trajectories as piecewise linear curves on the projected map plane and combines delay information with static schedule data to provide a visualization that is both close to reality and robust against missing real-time updates. We discussed the advantages of the approach compared to periodically updated GPS positions and developed a client/server system consisting of TrajServ, a partial trajectory HTTP server written in C++, and TRAVIC, a JavaScript client consisting of a high-performance vector layer for map services.

Evaluation showed that even in dense transportation networks, the average request times for TrajServ are very low (usually between 1 and 80 ms). Further tests showed that the request times stay the same if the dataset grows in both the number of trajectories and covered area. In Section 6.2, we demonstrated that client performance is also very good, resulting in smooth vehicle movements for almost every scenario, on different browsers. The only case where we could measure significant performance problems was during the visualization of New York City in Internet Explorer 11. Both TrajServ and TRAVIC are stable. Their source codes have been published under GPL v2 and are available to download[1].

For large update intervals, the problem of asynchronous delay information poses a problem that has yet to be addressed. We proposed possible solutions to this issue in Section 6.3. In addition, future work will include further optimization of the server performance (especially memory usage) as well as the adding of additional features. We also plan a route planner integration.

During the development process, we noticed the lack of proper methods to extract vehicle routes from geospatial data. In Section 7.2, we discussed two general strategies to solve the problem of associating trajectories that only consist of station coordinates to exact geospatial paths: map matching and shortest-path iteration. Further investigations will concentrate on how to apply these approaches programmatically to huge datasets. We plan to implement an algorithm that uses global map matching on OpenStreetMap data to find vehicle routes for GTFS feeds without a `shapes.txt` file.

[1] `https://github.com/patrickbr/trajserver`
`https://github.com/patrickbr/TRAVIC`

At the outset of this thesis, we planned to provide a toolset able to create live maps for arbitrary public transit feeds in the GTFS format. With TrajServ and TRAVIC, it is not only possible to do real-time public transit visualizations for single cities, but for entire areas or countries. With the increasing availability of static and real-time GTFS feeds, we hope that in the future, TrajServ and TRAVIC will provide the framework for a live map that covers the entire world.

A. TrajServ Request Parameters and JSON Output

This appendix serves as a manual to TrajServ's request parameters and to the format of their JSON output. TrajServ knows four basic request commands: `vehiclepos`, `trajectories`, `trajectory` and `trajstations`. Request parameters must be given as GET-parameters. Each request can have optional fields **`cb`** and **`rid`**. The callback **`cb`** can be used to pad the JSON with a callback function to get a JSONP output that can be accessed across domains. The request ID **`rid`** is passed through unchanged to the output and helps to distinguish different request answers. If no callback function was specified, the output will be raw JSON.

/vehiclepos

This is the most basic command. It outputs the current positions of all vehicles within a request rectangle. This is a mapping of a `trajectories` request with `btime`=`etime`. Parameters are:

`swx=`⟨*integer*⟩, **`swy=`**⟨*integer*⟩, **`nex=`**⟨*integer*⟩, **`ney=`**⟨*integer*⟩

The lower left (`swx`, `swy`) and the upper right (`nex`, `ney`) corner of the request rectangle, given in absolute projected coordinates (**not** as latitude/longitude).

`orx=`⟨*integer*⟩, **`ory=`**⟨*integer*⟩

Coordinates will be outputted relative to (`orx`, `ory`). Usually, this is the upper left corner of the clients current viewbox.

`date=`⟨*YYYY:MM:DD*⟩

The request date.

`time=`⟨*HH:MM:SS*⟩

A single request time.

`z=`⟨*1-32*⟩

The current zoom level.

The server will output a list of all vehicles currently moving through the request rectangle, containing their IDs, types, human readable names, colors and positions.

Field `rid` holds the request ID, `tz` holds all timezones currently displayed and `a` contains a list of vehicle positions (`p`) along with their parameters. `<TIMEZONE_OFFSET>` is in seconds.

```
{
  "rid": <REQUEST_ID>,
  "tz": [
      {
        "c": <TIMEZONE_CODE>,
        "os": <TIMEZONE_OFFSET>
      }*
  ],
  "a": [
      {
        "id": <TRAJECTORY_ID>,
        "t": <0 - 7>,
        "c": <HEX COLOR>,
        "tc": <HEX COLOR>,
        "hs": <HEADSIGN>,
        "sn": <SHORT_NAME>,
        "ln": <LONG_NAME>,
        "p":  [<X>, <Y>]
      }*
  ]}
```

/trajectories

This is the command for spatiotemporal requests as described in Section 2.2.2. It allows to get all clipped vehicle trajectories that cross a rectangle in a certain timespan. Parameters are essentially the same as with `vehiclepos`, except for `time`, which is replaced by `btime` and `etime`:

`btime=`⟨*HH:MM:SS*⟩

The begin time of this request's timespan.

`etime=`⟨*HH:MM:SS*⟩

The end time of this request's timespan.

The answer format resembles that of `vehiclepos`, but instead of positions, partial trajectories are outputted for each vehicle (see the JSON output of `trajectory` below for a specification of `<PARTIAL_TRAJECTORY_WRAP>`).

```
{
  "rid": <REQUEST_ID>,
  "tz": [
      {
        "c": <TIMEZONE_CODE>,
        "os": <TIMEZONE_OFFSET>
      }*
  ],
  "a": [
      <PARTIAL_TRAJECTORY_WRAP>*
  ]}
```

/trajectory

A single trajectory version of `vehiclepos`. This can for example be used to display the path a specific vehicle takes. Parameters are the same is for `trajectories`, except for

id=⟨*integer*⟩

The trajectory ID.

```
{
  "rid": <REQUEST_ID>,
  "id": <TRAJECTORY_ID>,
  "t": <0 - 7>,
  "c": <HEX COLOR>,
  "tc": <HEX COLOR>,
  "hs": <HEADSIGN>,
  "sn": <SHORT_NAME>,
  "ln": <LONG_NAME>,
  "pts": [
        [
          <SPATIOTEMPORAL_WAYPOINT>*
        ]*
  ]
}
```

The server outputs a single `<PARTIAL_TRAJECTORY_WRAP>` describing a single trajectory's way through a spatiotemporal bounding box. Field `id` is the (internal) trip ID, `t` is the GTFS type of the vehicle, `c` and `tc` are color and text color of the vehicle, `hs` is the headsign (for example "Amsterdam Centraal"), `sn` is the vehicle's short name (often the train ID) and `ln` is the long name. Field `pts` holds partial trajectories.

`<SPATIOTEMPORAL_WAYPOINT>` is either a timestamped waypoint

```
{
  "p": <WAYPOINT>,
  "at": <ARRIVAL_TIME>,
  "dt": <DEPARTURE_TIME>,
  "ad": <ARRIVAL_DELAY>,
  "dd": <DEPARTURE_DELAY>,
  "sid": <STATION_ID>
}
```

where `p` is a normal waypoint, `at` and `dt` are arrival and departure times, given as POSIX timestamps (UTC), `ad` and `dd` are delays, `sid` is an optional field holding station IDs. `<SPATIOTEMPORAL WAYPOINT>` can also just be a raw normal waypoint `[<INTEGER, INTEGER>]`, where the first field is the x-coordinate, the second field the y-coordinate.

/trajstations

Outputs the complete list of stations of a trajectory, for example to display the vehicle's schedule. This command only takes a single parameter:

id=⟨*integer*⟩

The trajectory ID.

```
{
  "rid": <REQUEST_ID>,
  "id": <TRAJECTORY_ID>,
  "t": <0 - 7>,
  "c": <HEX COLOR>,
  "tc": <HEX COLOR>,
  "hs": <HEADSIGN>,
  "sn": <SHORT_NAME>,
  "ln": <LONG_NAME>,
  "tt": {
        "t": <BITMAP AS INTEGER>,
        "n": [
          <DATE_OB>,
        ],
        "p": [
          <DATE_OB>
        ]
  },
  "sts": [
        {
          "p": [<INTEGER>, <INTEGER>],
          "at": <ARRIVAL_TIME>,
          "dt": <DEPARTURE_TIME>,
          "ad": <ARRIVAL_DELAY>,
          "dd": <DEPARTURE_DELAY>m
          "sid": <STATION_ID>,
          "n": <STATION_NAME>
        }*
  ]
}
```

The list of stations is outputted to `sts`, the service dates of this trajectory are outputted as a bitmap to `<BITMAP AS INTEGER>`. Positive service date exceptions are stored in `tt.p`, negative service date exceptions in `tt.n`.

List of Figures

List of Tables

Bibliography

[1] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, Christos Zaroliagis. *Efficient Models for Timetable Information in Public Transportation Systems.* In ACM Journal of Experimental Algorithmics, Volume 12, 2008.

[2] Matthias Müller-Hannemann, Franz Schulz, Dorothea Wagner, Christos Zaroliagis. *Timetable Information: Models and Algorithms.* In Proceedings of the 4th Workshop on Algorithmic Methods for Railway Optimization (ATMOS 2004), pp. 67-90, 2004.

[3] *General Transit Feed Specification.*
`https://developers.google.com/transit/gtfs/reference`

[4] *GTFS-realtime Reference.*
`https://developers.google.com/transit/gtfs-realtime/reference`

[5] *UML Diagram of the General Transit Feed Specification.*
`http://www.google.com/help/hc/images/transitpartners_1106431_objecttablelarge_en.gif`

[6] *OpenStreetMap Wiki.*
`http://http://wiki.openstreetmap.org`

[7] Aaron Antrim and Sean J. Barbeau. *The Many Uses of GTFS Data - Opening the Door to Transit and Multimodal Applications.* In Proceedings of the 2013 annual meeting of the Intelligent Transportation Society of America, 2013.

[8] *GTFS Data Exchange.*
`http://www.gtfs-data-exchange.com`

[9] Antonin Guttman. *R-Trees: A Dynamic Index Structure for Spatial Searching.* In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data - SIGMOD '84, p. 47, 1984.

[10] Mario A. Nascimento, Jefferson R. O. Silva and Yannis Theodoridis. *Evaluation of Access Structures for Discretely Moving Points.* In Proceedings of the International Workshop on Spatio-Temporal Database Management, pp. 171-188, 1999.

[11] John P. Snyder. *Map projections - A working manual.* U.S Geological Survey Professional Paper 1395, pp. 3-47, Washington, 1987.

[12] *Department of Defense World Geodetic System 1984, Its Definition and Relationships With Local Geodetic Systems.* NIMA Technical Report TR8350.2. 3rd

Edition, National Geospatial-Intelligence Agency, 2000.

[13] Eric W. Weisstein. *Great Circle.* `http://mathworld.wolfram.com/GreatCircle.html`

[14] *Great circle distance between 2 points.* U.S. Census Bureau Geographic Information Systems FAQ. `http://www.movable-type.co.uk/scripts/gis-faq-5.1.html`

[15] Bob Sproull and William M. Newman. *Principles of Interactive Computer Graphics.* International Edition, pp. 3-47, McGraw-Hill Education, 1973.

[16] *Google Protocol Buffers Reference.* `https://developers.google.com/protocol-buffers/docs/overview`

[17] *Freiburg Wiwilibrücke - Radfahrende im Querschnitt.* `http://fr-wiwili.visio-tools.com`

[18] Straßenverkehrszentrale Baden-Württemberg. *Automatische Straßenverkehrszählungen in Baden-Württemberg. Ergebnisse Juni 2013*, p. 13. Regierungspräsidium Tübingen - Landesstelle für Straßentechnik, 2013. `http://www.svz-bw.de/fileadmin/verkehrszaehlung/dz/2013/rpt-95-vz-2013-06.pdf`

[19] David Douglas, Thomas Peucker. *Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature.* In The Canadian Cartographer, Vol. 10, No. 2, pp. 112-122.

[20] Subhash Suri. *A Linear Time Algorithm with Minimum Link Paths Inside a Simple Polygon.* In Computer Vision, Graphics, and Image Processing, Vol. 35, No. 1, pp. 99-110, 1986.

[21] Adrian Batzill, Patrick Brosi, Susanne Eichel, Niklas Meinzer. *Implementation Description of a Multimodal Route Planner for the Freiburg Area.* Team project at the University of Freiburg, 2013. `http://panarea.informatik.uni-freiburg.de/routeplanner/projectinfo/`

[22] Sotiris Brakatsoulas, Dieter Pfoser, Randall Salas, Carola Wenk. *On map-matching vehicle tracking data.* In Proceedings of the 31st International Conference on Very Large Data Bases, pp. 853-864. VLDB Endowment, 2005.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, den 6. März 2014

Patrick Brosi